EGD URDU LOCALIZATION PROJECT

Parser Detailed Design (MT)

Oct 31, 2005

CENTER FOR RESEARCH IN URDU LANGUAGE PROCESSING
NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES, LAHORE
PAKISTAN

Table of Contents

Revision History

Reference No: EGD/MT/

Name	Change Date	Version	Description of Changes
MT Team	Oct 31, 2005		Created
Huda	Feb 13, 2006		Adding complete FS design

1. High Level Design

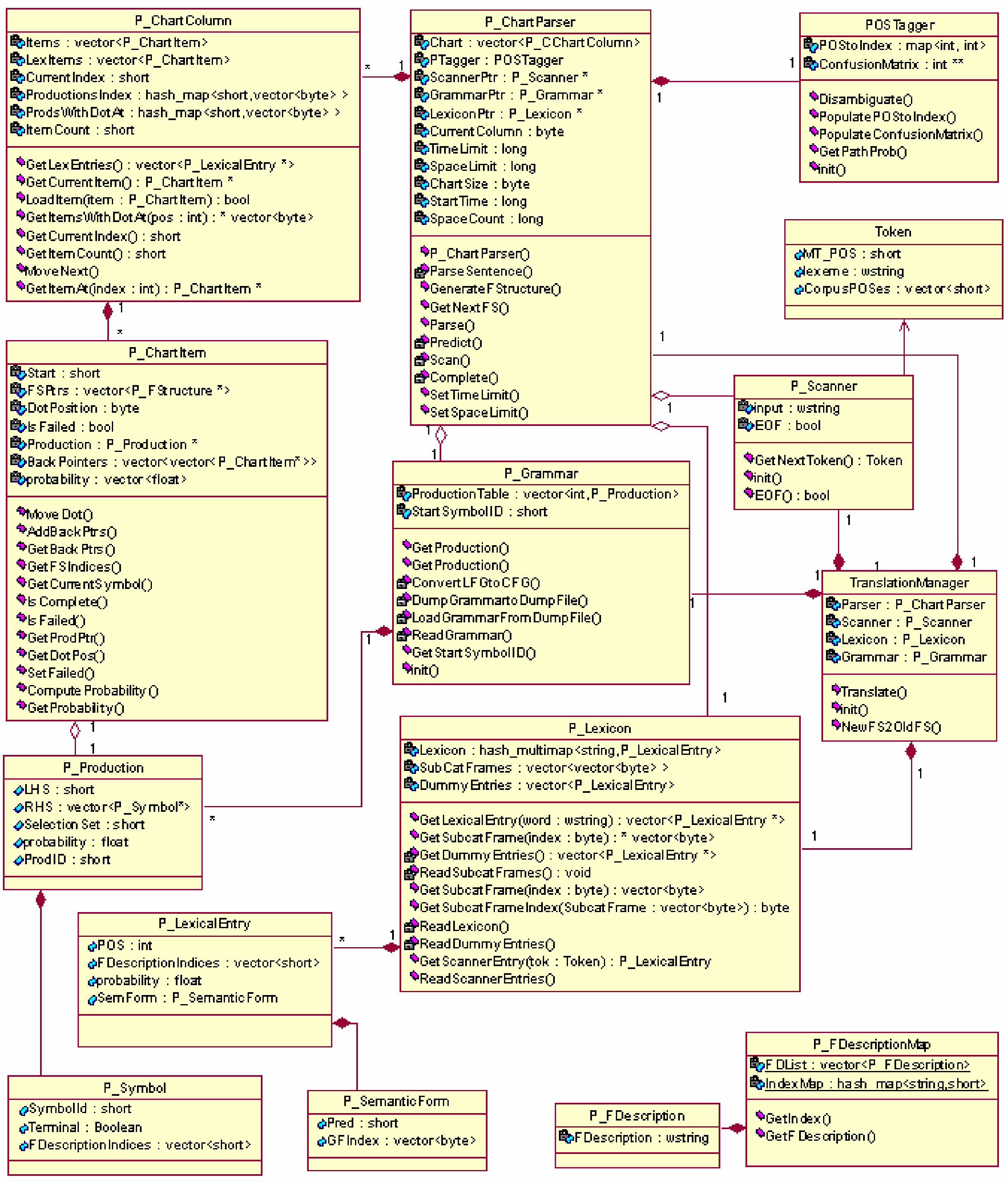
1.1. System Architecture

Description:

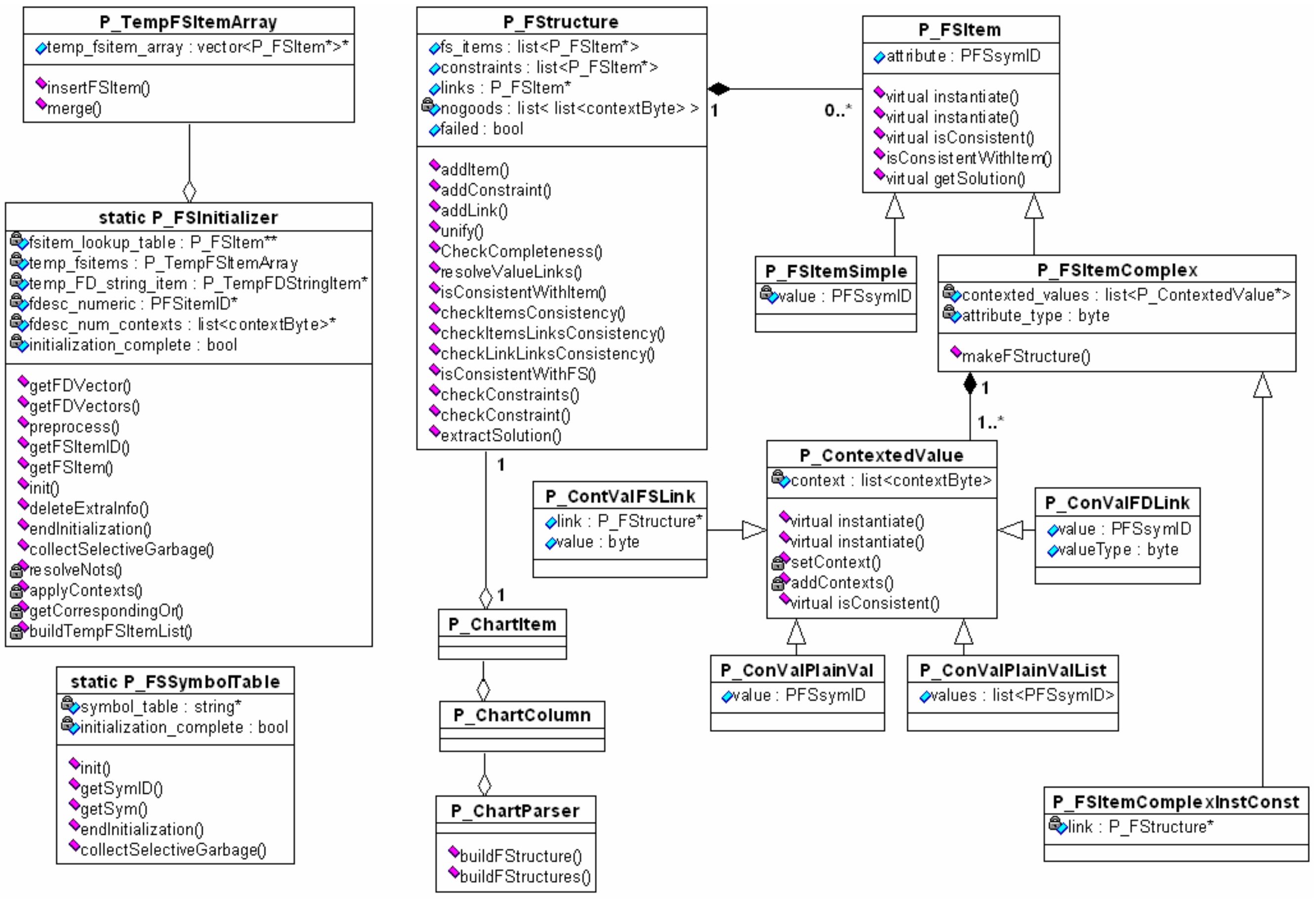
1.2. List of Files:

File	Description	Type	Format
ELexicon.lex	It contains raw lexicon entries of English Lexicon.	Text	Unicode
EGRules.gr		Text	Unicode
EGMacros		Text	Unicode
Egrammar		Text	Unicode
Egrammar.dmp		Binary	
Elexicon.dmp		Binary	
POSes.txt		Text	Unicode
BigramProb.txt		Binary	
TrigramProb.txt		Binary	
SubcatFrames.txt		Text	Unicode
DummyLex.lex			
fsTypeAttribute.txt	Contains list of attribute names that require an f-structure as their value, e.g., “SUBJ”, “XCOMP” etc.	Text	ANSI

2. Detailed Design



The following is the class diagram relevant to f-structure building.



2.1. Classes Description

P_SemanticForm

```
struct P_SemanticForm
{
      //    Attributes:
```

Name	Type	Description
Pred	Short	The “Pred” is the Predicate of the Lexical Entry having the index of the Symbol table
GFIndex	vector<byte>	This is the vector of indices of all subcat frame vectors attached with the Lexical entry. These indices correspond to the indices of SubcatFrames vector of P_Lexicon

```
};
```

P_LexicalEntry

```
struct P_LexiconEntry
{
      //    Attributes:
```

Name	Type	Description
POS	int	A number representing the POS of the Lexical Entry. The possible values of POS are in power of 2. for each POS, this power is the position number of that POS in POSes.txt.
FDescriptionIndices	vector<short>	Indices of FDescriptions attached with the Lexical Entry. Actual FDescriptions against these indices can be retrieved from P_FDescriptionMap class.
probability	Float	The probability of the Lex Entry as “pos”
SemForm	P_SemanticForm	Semantic Form of the Lexical Entry (Predicate + Grammatical Function)

```
};
```

P_Symbol

```
struct P_CSymbol
{

      //    Attributes:
```

Name	Type	Description
SymbolId	short	This is the Symbol Id
IsTerminal	bool	This Symbol is the Terminal or NonTerminal
FDescriptionIndices	vector<short>	Functional Description of each Symbol of the RHS

```
};
```

P_Production

```
struct P_CProduction
{
```

// Attributes:

Name	Type	Description
LHS	short LHS	Production LHS Symbol
RHS	vector<P_CSymbol*> RHS	Production RHS Symbols
SelectionSet	short SelectionSet	Bit Representation of Selection Set. For each POS in the selection set of the production, corresponding bit is turned on in the SelectionSet variable. (recall that POSes’ values are in Power of two)

```
};
```

Token

```
struct Token
{
```

// Attributes:

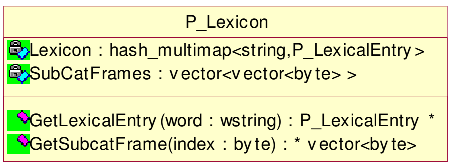
Name	Type	Description
MT_POS	int	
lexeme	string	
CorpusPOSes	vector<short>	

```
};
```

P_FDescription

P_Scanner

P_Lexicon


P_Lexicon
Lexicon : hash_multimap<string,P_LexicalEntry >
SubCatFrames : vector<vector<byte> >
GetLexicalEntry (word : wstring) : P_LexicalEntry *
GetSubcatFrame(index : byte) : * vector<byte>

Attributes

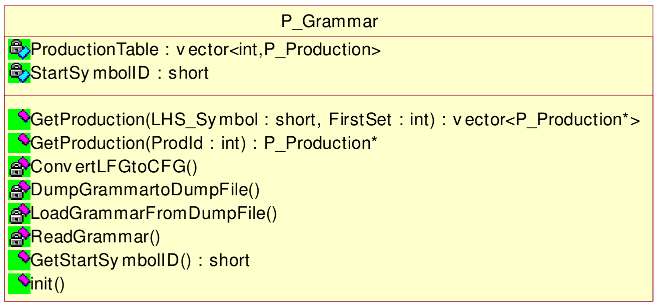
Name	Type	Description
Lexicon	hash_multimap<wstring,P_LexicalEntry>	The Lexical Entries Hash_Map hashed on the lexeme
SubCatFrames	vector<vector<byte> >	All the possible Subcat frames vector: <SUBJ,OBJ>

Functions

`GetLexicalEntry(word : wstring) : vector<P_LexicalEntry *>`
It returns the list of the lexical entries corresponds to the "word"

`GetSubcatFrame(index : byte) : vector<byte>*`
Return a list of the Subcat frames corresponds to the "index"

P_Grammar


P_Grammar
ProductionTable : vector<int,P_Production>
StartSymbolID : short
GetProduction(LHS_Symbol : short, FirstSet : int) : vector<P_CProduction*>
GetProduction(ProdId : int) : P_Production*
ConvertLFGtoCFG()
DumpGrammartoDumpFile()
LoadGrammarFromDumpFile()
ReadGrammar()
GetStartSymbolID() : short
init()

It is the data structure that contains the information of All Grammar Rules. It also provides the functionality to convert the grammar rules to CFG format.

Attributes:

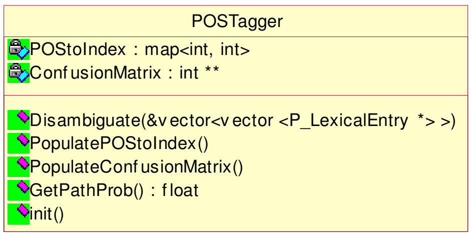
Name	Type
ProductionTable	multimap<int,P_CProduction*>
StartId	Short //'S'

Operations:

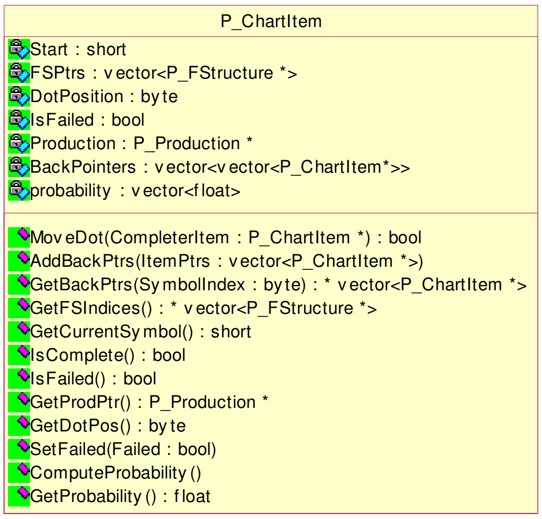
Name	Signature
GetProduction	vector<P-CProduction*> GetProduction(LHS_Symbol, int POS)
GetStartId	Short GetStartId()
ReadLFGGrammar	void ReadLFGGrammar()
ConvertLFG2CFG	ConvertLFG2CFG (String InputFName, String OutputFName)
GetProductionList	GetProductionList (String Lhs)
GetProduction	P-CProduction* GetProduction(int ProdId)
WriteCFG	void WriteCFG()
MergeProductions	void MergeProductions()
LoadGrammarFromDumpFile	void LoadGrammarFromDumpFile(string Path)

```
Cases:

   ________________________________________________________________________

1: OPTIONAL    (   ) and OR |

   A -> B ( D | E ) F
   // conversions:
    A -> B X2 F
    A -> B F
    X2 -> D
    X2 -> E
   ________________________________________________________________________

2: STAR ( * )

    A -> B [ D | E ]* F
   // conversions:
    A -> B X2 F
    A -> B F
    X2 -> [D | E] X2
    X2 -> [D | E]
   ________________________________________________________________________
3: PLUS  (+)

A -> B [ D | E ]+ F

// conversions:

A -> B X2 F
X2 -> [D | E] X2
X2 -> [D | E]

   ________________________________________________________________________

4: No of times ( # )

A -> B [ D % E ]#3 F

// conversions:
A -> B X2 F
A -> B X2 X2 F
A -> B X2 X2 X2 F
X2-> [D % E]

   ________________________________________________________________________

5: Shuffle ( % )

   X -> [A%B]
// conversions:
   X -> A B
   X -> B A

*/
```

POSTagger


POSTagger
POStoIndex : map<int, int>
ConfusionMatrix : int **
Disambiguate(&vector<vector <P_LexicalEntry *> >)
PopulatePOStoIndex()
PopulateConfusionMatrix()
GetPathProb() : float
init()

P_ChartItem



```
P_ChartItem
Start : short
FSPtrs : vector<P_FStructure *>
DotPosition : byte
IsFailed : bool
Production : P_Production *
BackPointers : vector<vector<P_ChartItem*>>
probability : vector<float>

MoveDot(CompleterItem : P_ChartItem *) : bool
AddBackPtrs(ItemPtrs : vector<P_ChartItem *>)
GetBackPtrs(SymbolIndex : byte) : * vector<P_ChartItem *>
GetFSIndices() : * vector<P_FStructure *>
GetCurrentSymbol() : short
IsComplete() : bool
IsFailed() : bool
GetProdPtr() : P_Production *
GetDotPos() : byte
SetFailed(Failed : bool)
ComputeProbability()
GetProbability() : float
```

Attributes

Name	Type	Description
Start	short	Index of the chart column where the current P_ChartItem was first placed
FSPtrs	vector<P_FStructure *>	Pointers of P_FStructure corresponding to the each possible tree having current P_ChartItem as root.
DotPosition	byte	Indicates the position of dot in the production of current P_ChartItem. It can have values from 0 to Number of right-hand-side symbols in the production.
IsFailed	bool	Boolean value indicating whether the subtree having root at current chart item was failed during F-Structure building or not.
Production	P_Production *	Pointer to a production in Grammar
BackPointers	vector<vector<P_ChartItem *> >	Conatains a separate vector of back pointers for each alternative subtree with root on current chart item. For a single subtree the corresponding vector contains a pointer of P_ChartItem for every right-hand-side symbol of production, pointing to that completed chart item that moved the dot over the symbol.
Probability	vector<float>	Contains a probability corresponding to every subtree rooted at current chart item

Functions

`MoveDot(CompleterItem : P_ChartItem *) : bool`
Moves dot, adds one to the DotPosition, if possible. If the Dot can be moved then it first adds **CompleterItem** pointer to all the BackPointers vectors on their Dot Position Index.

`AddBackPtrs(ItemPtrs : vector<P_ChartItem *>)`
Adds a new vector of P_ChartItem pointers (ItemPtrs) to the BackPointers vector. Back pointers are added from 0th symbol uptill the last element before DotPosition. Hence if the DotPosition is 0, no BackPointers are added.

`GetBackPtrs() : vector<vector<P_ChartItem *> >`
Returns BackPointers.

`GetFSIndices() : * vector<P_FStructure *>`
Returns FSPtrs.

`GetCurrentSymbol() : short`
From among the right-hand-side symbols of the production attached with current chart item, the id of the one on the DotPosition index is returned.

`IsComplete() : bool`
Returns true if the DotPosition is equaql to the number of right-hand-side symbols in the production attached with the chart item.

`IsFailed() : bool`
Returns IsFailed.

`GetProdPtr() : P_Production *`
Returns Production.

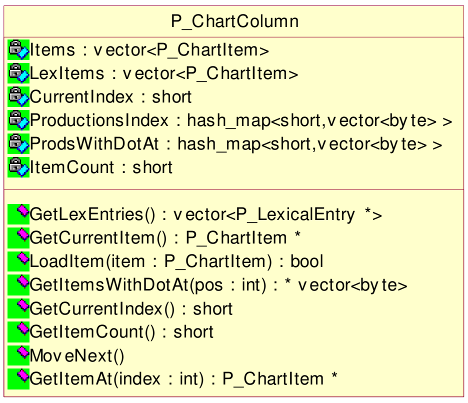
`GetDotPos() : byte`
Returns DotPsition.

`SetFailed(Failed : bool)`
Sets IsFailed equal to **`Failed`**.

`ComputeProbability()`
For each vector of back pointers it adds the probabilities of all the chart items pointed by the back pointers and then adds the probability of the Production to it. The sum is then assigned at proper index (corresponding to that subtree whose back pointers were used for calculating probability) in probability vector.

`GetProbability() : float`
Returns Probability.

P_ChartColumn


P_ChartColumn
Items : vector<P_ChartItem>
LexItems : vector<P_ChartItem>
CurrentIndex : short
ProductionsIndex : hash_map<short,vector<byte> >
ProdsWithDotAt : hash_map<short,vector<byte> >
ItemCount : short
GetLexEntries() : vector<P_LexicalEntry *>
GetCurrentItem() : P_ChartItem *
LoadItem(item : P_ChartItem) : bool
GetItemsWithDotAt(pos : int) : * vector<byte>
GetCurrentIndex() : short
GetItemCount() : short
MoveNext()
GetItemAt(index : int) : P_ChartItem *

Attributes

Name	Type	Description
Items	vector<P_ChartItem>	Vector od items contained in the column.
LexItems	vector<P_ ChartItem>	Chart items corresponding to the lexical entries of the word attached with the chart column.
CurrentIndex	short	Index of the chart item being currently processed.
ProductionsIndex	hash_map<short,vector<byte> >	A hash_map that has production id as key and against this key it contains the indices of all those chart items which contain the pointer of the grammar production having that production id.
ProdsWithDotAt	hash_map<short,vector<byte> >	A hash_map that has P_Symbol as key and against this key it contains the indices of all those chart items which contain the pointer of the grammar production having that symbol on DotPosition on the right-hand-side of production.
ItemCount	short	Total number of items in the column.

Functions

`GetLexEntries() : vector<P_LexicalEntry *>`
gives the vector of pointers of P_LexicalEntry attached with column

`GetCurrentItem() : P_ChartItem *`
it returns the chart item locate at the CurrentIndex

`LoadItem(item : P_ChartItem) : bool`
Loads a chart item in the chart if it is not already there. It finds the item in the chart column using ProductionsIndex hash_map. If the chart item with exactly the same production (with same dot position) is already there and DotPosition is not 0 then it adds the back pointers' vector of new comer chart item to the vector of back pointers' vectors of existing chart item

`GetItemsWithDotAt(symb : short) : * vector<byte>`
Returns the indices of all those chart items whose production symbol at DotPosition is equal to the **symb**, this is done by hashing into the 'ProdsWithDotAt' hash map.

`GetCurrentIndex() : short`
Return CurrentIndex

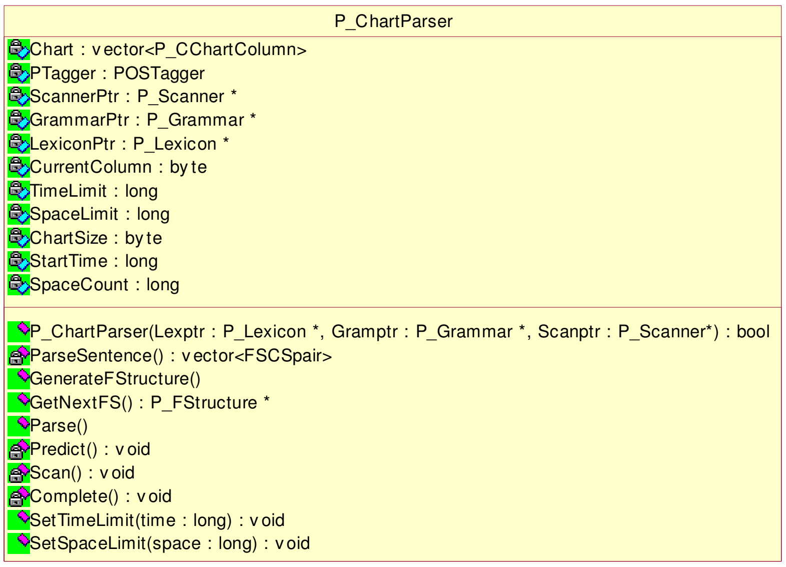
`GetItemCount() : short`

Returns ItemCount

`MoveNext()`
Increments CurrentIndex

`GetItemAt(index : int) : P_ChartItem *`
Returns the pointer of the chart item located at **index**.

P_ChartParser



P_ChartParser

Chart : vector<P_CChartColumn>
PTagger : POSTagger
ScannerPtr : P_Scanner *
GrammarPtr : P_Grammar *
LexiconPtr : P_Lexicon *
CurrentColumn : byte
TimeLimit : long
SpaceLimit : long
ChartSize : byte
StartTime : long
SpaceCount : long

P_ChartParser(Lexptr : P_Lexicon *, Gramptr : P_Grammar *, Scanptr : P_Scanner*) : bool
ParseSentence() : vector<FSCSpair>
GenerateFStructure()
GetNextFS() : P_FStructure *
Parse()
Predict() : void
Scan() : void
Complete() : void
SetTimeLimit(time : long) : void
SetSpaceLimit(space : long) : void

Attributes

Name	Type	Description
Chart	Vector<P_ChartColumn>	
PTagger	POSTagger	
ScannerPtr	P_Scanner	
GrammarPtr	P_Grammar	
LexiconPtr	P_Lexicon	
CurrentColumn	Byte	
TimeLimit	Long	
SpaceLimit	Long	
ChartSize	Byte	
StartTime	Long	
SpaceCount	long	

Functions

`P_ChartParser(Lexptr : P_Lexicon *, Gramptr : P_Grammar *, Scanptr : P_Scanner*) : bool`
Initializes LexiconPtr, GrammaerPtr, and ScannerPtr to **Lexptr**, **Gramptr** and **Scanptr**.

`Parse() : void`
This function gets the tokens of a complete sentence from ScannerPtr and for each token retrieves its lex entries from LexiconPtr. Then it sends these entries to PTagger for POS-disambiguation. It resizes the Chart to the size equal to sentense-length+1. To each column of the chart it assigns its corresponding lex entries and then calls ParseSentence() function.

`ParseSentence() : vector<FSCSpair>`
This function is mainly responsible for parsing. It puts all productions of ‘S’ in the 0th chart column and begins parsing. During parsing it traverses all chart columns using CurrentColumn variable and for each column it sequentially traverses all its items. For each item it checks if the item is completed, if so then it calls Complete()

otherwise it checks the current symbol of the production attached to the item. If the symbol is terminal then it calls Scan() otherwise it calls Predict(). During traversal of chart column, in each iteration it checks the times elapsed to for and the memory occupied so far and if any of them exceeds its limit, it releases the memory allocated during parsing and exits the function returning Null value.

`Predict() : void`
This function is called when the current symbol of the current chart item is a non terminal. It retrieves all those production of the non-terminal symbol whose first set contains the POS of any of the lex entries attached to the current chart item. It places each of these production in a new chart item in the current chart column. It uses LoadItem() function of P_ChartColumn.

`Scan() : void`
Scan() is called when the current symbol of the current chart column is a terminal. It compares the terminal symbol with the POSes of the lex entries of the current chart column and if it matches with any then It makes a copy of the item, Moves its dot and puts it in the next column of the chart. It uses MoveDot function of the P_ChartItem.

`Complete() : void`
This function is called if the current chart item contains completed production (whose DotPosition is at the ends of the RHS of production). It goes to the column where the production was first introduced by the predictor and from that column it retrieves all those items whose current symbol matched the LHS of the production of the completed item (these are the items who actually introduced the production). It moves the dot of all these production by one step. it adds the pointer of completed item to the back pointers of the item and then loads this item to the current column of the chart using LoadItem() function.

`GenerateFStructure()`

`GetNextFS() : P_FStructure *`

`SetTimeLimit(long time) : void`

`SetSpaceLimit(long space) : void`

`bool buildFStructure(P_ChartItem* chart_item)`
The dummy (S') starting production (which leads to 1 or more S productions) from the grammar is sent as parameter (`chart_item`) to start the f-structure building process. This consists of 3 main things, 1) allocate space for a new `P_Fstructure`, set the chart_items's `FSPtr` to it, and then send it as a parameter to the recursive function `builFStructures()` so that the complete f-structure corresponding to the sentence can be built; 2) checks the constrainst of the final f-structure that is built using the function `checkConstarints()`; and 3) check the f-structure for completeness and coherence using the function `CheckCompleteness()`.

`bool buildFStructures(P_ChartItem* chart_item, P_FStructure* f_structure, contextByte& context_offset)`
This function builds the complete f-structure recursively. To do this 2 main steps are required for each RHS symbol of `chart_item`:

1) by inspecting the back-pointers for the current symbol figure out the set of down-arrow values for each RHS symbol fsitem; to do this, for each back-pointer that doesn't already have an f-structure assigned to it, call `buildFStructures()`.

2) instantiate the fs items of the RHS symbol and put them in the f-structure.

The pseudo-code of how this is done is as follows:

```
outgoing_offset = context_offset;
outgoing_offset = outgoing_offset + production->MAXCONTEXT;
```

```
for (all RHS production symbols)
{
      firstly:
      collect all unique back-pointers for RHS_symbol in curr_sym_bkptrs;
      (back-pointers for an RHS symbol will be in a single column)

      if (RHS_symbol is not a terminal)
      {
            if (multiple back-pointers)
            {
                  prepare the contexts that will arise due to multiple back-pointers;
                  adjust outgoing_context accordingly;

                  for (all curr_sym_bkptrs)
                  {
                        if (curr_sym_bkptr has no FSPtr)
                        {
                              set it's FSPtr to fs_RHS_symbol;
                              make recursive call to build fs_RHS_symbol;
                              buildFStructures(curr_sym_bkptr, fs_RHS_symbol,
                              outgoing_offset);
                        }
                        else
                              set fs_RHS_symbol to the FSPtr;


                        prepare link_value to be used as a reference (i.e., a down-
                        arrow) to the curr_sym_bkptr as follows:

                        link_value->context = prepared context for curr_sym_bkptr
                        link_value->link = fs_RHS_symbol;
                        link_value->value = 0;
                        link_value->value_type = DOWNARROWVAL;

                        add link_value to down_arrow_values;
                  }
            }

            else (i.e. single back-pointer)
            {
                  if (curr_sym_bkptr has no FSPtr)
                        {
                              set it's FSPtr to fs_RHS_symbol;
                              make recursive call to build fs_RHS_symbol;
                              buildFStructures(curr_sym_bkptr, fs_RHS_symbol,
                              outgoing_offset);
                        }
                  else
                        set fs_RHS_symbol to the FSPtr;

            }
      }

      else (i.e., lexical items have been reached)
      {
            lexical item f-structures will be built here (end of recursion);
            get lex_entry from curr_sym_bkptrs;
```

```
        if (lex_entry has no FSPtr)
        {
              for (all fsitems of lex_entry)
              {
                    inst_fsitem = fs_item->instantiate(outgoing_offset,
                    f_structure, NULL);

                    determine what type inst_fsitem is:
                    fsitem_for_list = inst_fsitem->getItemForFS();
                    constraint_for_list = inst_fsitem->getConstraintForFS();
                    link_for_list = inst_fsitem->getLinkForFS();

                    out of the three function calls above only one will return a
                    non-NULL value;

                    if (fsitem_for_list)
                          fs_RHS_symbol->addItem(fsitem_for_list);

                    if (constraint_for_list)
                    {
                          to complete constraint instantiation, the up-arrow /
                          down-arrow next to the down-arrow has to be determined:
                          inst_const = new
                          P_FSItemComplexInstConst(constraint_for_list,
                          f_structure, fs_RHS_symbol);
                          constraint_for_list = inst_const;
                          fs_RHS_symbol->addConstraint(constraint_for_list);
                    }

                    if (link_for_list)
                          fs_RHS_symbol->addLink(link_for_list);
              }

              if(lex_entry has a Pred)
              {
                    create pred_item;
                    fs_RHS_symbol->addItem(pred_item);

                    create index_item
                    fs_RHS_symbol->addItem(index_item);
              }

              outgoing_offset = outgoing_offset + lex_entry->MAXCONTEXT;

              fs_RHS_symbol->resolveValueLinks();
              fs_RHS_symbol->unify();
              lex_entry->FSPtr = fs_RHS_symbol;
        }
        else
              fs_RHS_symbol = lex_entry->FSPtr;
  }

  secondly:
  for (each RHS_symbol fsitem)
  {
        if (bkptr_size <= 1)
```

```
                  inst_fsitem = fs_item->instantiate(context_offset, f_structure,
                  fs_RHS_symbol);
            if (bkptr_size > 1)
                  inst_fsitem = fs_item->instantiate(context_offset, f_structure,
                  down_arrow_values);

            determine what type inst_fsitem is:
            fsitem_for_list = inst_fsitem->getItemForFS();
            constraint_for_list = inst_fsitem->getConstraintForFS();
            link_for_list = inst_fsitem->getLinkForFS();

            if (fsitem_for_list)
                  f_structure->addItem(fsitem_for_list);
            if (constraint_for_list)
            {
                  inst_const = new P_FSItemComplexInstConst(constraint_for_list,
                  f_structure, fs_RHS_symbol);
                  constraint_for_list = inst_const;
                  f_structure->addConstraint(constraint_for_list);
            }

            if (link_for_list)
                  f_structure->addLink(link_for_list);
      }
}

f_structure->resolveValueLinks();
f_structure->unify();

context_offset = outgoing_offset;
```

list< list<contextByte> > prepareBkPtrContextList(short no_of_contexts)
This is exactly the same as the context preparation function in `P_FSIntializer`, except that here the contexts are needed due to the presence of multiple backpointers for a `chart_item`.

bool vectorContainsItem(vector<P_ChartItem*>& item_vector, P_ChartItem* chart_item)
checks if the vector `item_vector` contains `chart_item`.

contextByte adjustBkPtrContextList(list< list<contextByte> >& prepared_context, contextByte start_from)
Adjust the backpointer contexts so that they start from `start_from` and returns the maximum value assigned.

P_FSInitializer

This class serves to convert the f-descriptions associated with the grammar productions and lexicon entries from their textual format into one that will be more efficient for processing purposes. Before the class is described in detail, the following few sections describes f-descriptions and the different forms they may take.

F-Descriptions

This section describes f-descriptions and the different forms that they may take.

Each lexical item in the lexicon and each right-hand symbol in the grammar is associated with a comma-separated list of f-descriptions, as shown in the examples below:

Grammar Production:

```
      MULTP -> card: ^=!, !NUM =c PL;
               mult: ^=!, !MULT_FORM =c 'times';.
```

Lexical Item:
```
      a:art, ^NUM = SG, ^DEF = NEG, ^NCOUNT = POS.
```

Operators in f-descriptions
There are two types of operators that may be used in an f-description, conditions and constraints. Condition operators include `=` and `$`, whereas constraint operators include `=c` and ≠.

Different types of f-descriptions
Two types of variation may occur within an f-description: 1) the presence or non-presence of arrows next to the LHS and the RHS, 2) a chain of items on the LHS or / and the RHS. These variations and their occurrence with the two types described earlier are as follows:

An exhaustive list of all possibilities is:
```
^LHS
!LHS
LHS
^RHS
!RHS
RHS
^ (on LHS)
^ (on RHS)
! (on LHS)
! (on RHS)

LHS1 LHS2 LHS3 … CON RHS1 RHS2 RHS3 …
```

Out of these the legal forms that an f-description with a condition operator can take are:

^LHS
~~!LHS~~
~~LHS~~
~~^RHS~~
!RHS
RHS
^ (on LHS)
~~^ (on RHS)~~
~~! (on LHS)~~
! (on RHS)

LHS1 LHS2 LHS3 … COND RHS1 RHS2 RHS3 …

The legal forms for an f-description with a constraint operator are:

^LHS
!LHS
~~LHS~~
^RHS
!RHS
RHS
^ (on LHS)
^ (on RHS)
! (on LHS)
! (on RHS)

LHS1 LHS2 LHS3 … CONS RHS1 RHS2 RHS3 …

Now the following sections will describe the attributes and functions used by `P_FSInitializer`.

Attributes

static P_FSItem fsitem_lookup_table**

This is a look-up table to keep all the unique f-descriptions as they're read off the lexicon and the grammar. They are stored in a generic manner so that they can be instantiated as required and put into f-structures. The ID for each fs item is (its index in the array + 1) (this is done so that no fs item has the ID 0). Due to this storage, a single entry in this table can be instantiated in different ways as and when required.

static PFSitemID fsitem_insertion_index

The index where the next newly created fs item is to be inserted.

static vector<PFSitemID>* fsitem_insert_acc

Serves to speed-up insertion of fs items into `fsitem_lookup_table`, the number of vectors will be equal to the number of unique symbols (SUBJ, NUM, SG etc) that are being used in the f-descriptions. As the fs items are inserted into `fsitem_lookup_table`, thier ID (PFSitemID) is inserted into `fsitem_insert_acc` such that it goes into the i_{th} vector where i is the ID of the symbol that is on the LHS of the f-decsription. So, the following f-description,

```
^SUBJ = !
```

will be inserted into the vector number that is the ID for `SUBJ`. In this way when a new fs item is constructed and has to be inserted `fsitem_insert_acc` aids in speeding up the search for an identical item (if an identical item exists it is not inserted).

static PFSitemID lookup_table_temp_index

After the lexicon and grammar have been loaded any more entries in `fsitem_lookup_table` are temporary and exist for that cycle only. `lookup_table_temp_index` indicates the index in `fsitem_lookup_table` where these temporary entries begin.

static P_TempFSItemArray temp_fsitems

Keeps temporary versions of `P_FSItem`s that need to be merged before they can be inserted into `fsitem_lookup_table`. It exists for the construction of a single `FDVector`.

static P_TempFDStringItem* temp_FD_string_item

Keeps single f-descriptions with their corresponding `P_FSItem` so that copies can be made to put in temp_fsitems (saves construction cost); exists for construction of all `FDVector`s. The structure for `P_TempFDStringItem` is as follows:

```
struct P_TempFDStringItem
{
      string f_desc_string;
      P_FSItem* f_item;
};
```

static PFSitemID temp_FD_si_index

Insertion index for `temp_FD_string_item`.

static hash_map<string, PFSitemID> dummy_hash

A dummy hash map that exists purely to facilitate insertion into `temp_FD_string_item`.

static PFSitemID* fdesc_numeric

To store an f-description in numeric form in order to perform manipulations like adding contexts and moving the ~ operator to the literals. The following numbers are used to represent the f-descriptions:

```
-1: [
-2: ]
```

`-3: |`
`-4: ,`
`-5: ~`
`-10:` end of the f-description
Other than these numbers an f-description itself is indicated by its ID in `temp_FD_string_item`.

`static int fdesc_num_counter`
To keep track of insertions into `fdesc_numeric`.

`static list<contextByte>* fdesc_num_contexts`
To build up the contexts such that they correspond to the f-description represented by `fdesc_numeric`. For each index of `fdesc_numeric` where an fs item is expected, the correspoding index of `fdesc_num_contexts` will have it's context list.

`static list< list<contextByte> > prod_context`
To store the context that applies to a production that has multiple f-descriptions.

`static contextByte max_assigned_context`
The maximum context value that is assigned in a production or lexical entry f-description.

`static list<string> fsTypeAttr`
Is a list of the symbols (if they are representing an attribute) used in f-descriptions that will require an f-structure as the value.

`static bool initialization_complete`
To indicate the end of the initializtion process after which entries in `fsitem_lookup_table` and the symbol table for f-descriptions will be temporary.

`static bool internal_call`
To check whether the call to `getFDVector()` is external or from `getFDVectors()`.

Functions
`P_FSInitializer(void)`
Constructor, does nothing because no object is meant to be instantiated.

`~P_FSInitializer(void)`
Destructor, does nothing because no object is meant to be instantiated.

`static vector<PFSitemID> getFDVector(string f_desc, contextByte &max_context)`
Processes an f-description (for a single grammar symbol) from it's raw form (factorizes it (not implemented yet) and applies the context), makes relevant entries in `fsitem_lookup_table` and returns a vector of numbers representing the f-description.

For example, in the following production:

```
n -> boy: ^pred = 'boy', ^GEND = M.
```

the argument would be the string "`^pred = 'boy', ^GEND = M`". This will be broken up in two and each expression will be processed and assigned a number. The vector that is returned will contain the assigned numbers for each expression. (note: f-descriptions of type `pred` will not be included in this scheme, it has only been used here as an example.)

`static vector< vector<PFSitemID> > getFDVectors(vector< vector<string> > f_descriptions, contextByte &max_context)`

Gets a whole set of f-descriptions that go with a production; the dimensions of the 2-dimensional vector `f_descriptions` are (number of RHS symbols * number of different f-descriptions for the production); `max_context` is not used here, it is assigned the maximum context number at the end so that it can be assigned to the the grammar production.

`static list< list<contextByte> > prepareProductionContextList(byte no_of_fdescs)`
Calculates the context that has to be applied if a production has multiple f-descriptions associated wiith it. For example, if there are six different f-descriptions associated with a production the context pattern that should be constructed is as follows:

```
-5 -4 -3 -2 -1
-5 -4 -3 -2  1
-5 -4 -3  2
-5 -4  3
-5  4
 5
```

`static bool preprocess(string f_desc)`
Creates `fdesc_numeric`.

`PFSitemID getFSItemID(P_FSItem* fs_item)`
If the item already exists in `fsitem_lookup_table`, it returns it's ID else it inserts it and returns it's ID.

`static P_FSItem* getFSItem(PFSitemID itemID)`
Returns a copy of the fs item that is at the (`itemID` – 1)th index. This is done so that no fs item is assigned the ID 0. The item at index 0 will get the ID 1, and when it is rquired the value at index (1 - 0) = 0 is retrieved.

`static void printDataStructures(ofstream &myfile)`
Prints `fsitem_lookup_table`. For testing purposes only.

`static void init(void)`
Serves as the constructor which will never be called because no object of `P_FSInitializer` is instantiated.

`static void deleteExtraInfo(void)`
Frees memory (after lexicon and grammar have been processed) allocated to `temp_FD_string_item` and assigns new memory (lesser than the previous) that is sufficient for working on a single cycle.

`static void endInitialization(void)`
To be called after the main initialization process (grammar and lexicon) has ended; after this all items sent to get their FDVector constructed will have their items and symbols stored temporarily (will be deleted as soon as they're useless - at the end of each sentence).

`static void collectSelectiveGarbage(void)`
Cleans up everything that is required for a single cycle only.

`static string getNextToken(string input, int &index)`
Takes the complete f-description associated with a symbol in a grammar production or a lexical entry and retrieves the single f-description item that starts at `index`. The following are examples of tokens that can be retrieved: `NUM =c SG`, `SUBJ NUM = PL`, `ADV_TYPE = {V_MOD, N_MOD}`. The following operators are also retrieved as a token: `[ ] , | ~`

`static string cleanUpToken(string token)`
Removes redundant spaces from a token. Spaces in complex expressions like `SUBJ NUM` are not removed.

`static string* reGetNextToken(string input)`

Retrieves components of an f-description assuming that redundant white spaces have been cleaned up, e.g., "SUBJ", "=c", "NUM".

static list<PFSsymID> reReGetNextToken(string input)
Retrieves individual value IDs from a list e.g., {SG, PL}.

static bool fdNumInsert(PFSitemID op)
Inserts an element into fdesc_numeric.

static PFSitemID getTempFDStringItemID(string fdesc)
Checks if the string already exists in temp_FD_string_item; if it does return the ID, else construct the P_FSItem, insert and then return the ID.

static P_FSItem* getTempFDStringItemCopy(PFSitemID item_id)
Returns a copy of the item corresponding to the ID.

static bool resolveNots(void)
Resolve the NOTs from the f-description in fdesc_numeric using DeMorgan's Law; expressions with upto 2 operands are handled, specifically the follwing three operations are carried out:

```
~[a|b] -> ~a,~b
~[a,b] -> [~a|~b]
~[a] -> ~a
```

static bool applyContexts(void)
Calculates what contexts are to be applied using fdesc_numeric. There are two types that may be applied, firstly that context which arises due to grammar productions that have multiple f-descriptions associated with them, secondly the context that arises when two expressions arebeing ORed.

static PFSitemID getCorrespondingOr(int index)
Used when applying contexts, index indicates the index of a [in fdesc_numeric. The function returns the index of the OR (|) corresponding to it (if there is one) else it returns a 0.

static bool buildTempFSItemList(void)
Traverses fdesc_numeric; gets each item (from temp_FD_string_item) as it was originally constructed; modifies item to accomodate the negations (change operator from ~ to !=) and the contexts; inserts final structure in temp_fsitems.

static void fixFormat(string& fdesc)
Changes the format of the incoming string so that: 1) all "&&"s are changed to ","s, 2) all "||"s are changed to "|"s.

static void updateFSItemInsertAcc(PFSsymID attr, PFSitemID insert_index)
Adds new entry to fsitem_insert_acc.

static void removeFromAcc(PFSsymID attr, PFSitemID insert_index)
Removes temporary entry from fsitem_insert_acc.

static void changeSetToOrs(string& fdesc)
Used to change substrings in f-descriptions of the form

```
ATTRIBUTE = {VALUE1, VALUE2...}
```

to

```
[[ATTRIBUTE = VALUE1 | VALUE2]|...]
```

Implemented but not used.

P_TempFSItemArray

The purpose of this class is to hold the temporary fs items that are built during FDVector construction and to help speed-up the building.

Attributes

`vector<P_FSItem*>* temp_fsitem_array`

Array of vectors of size equal to (the size of the fs symbol table + 1) – the +1 is necessary because symbol IDs start from 1. Each fsitem is inserted into that array index which is equal to the symbol ID of that fsitem's attribute (LHS), e.g. that for `GEND` in `GEND = SG`. This helps to merge items that have the same attribute.

Functions

`P_TempFSItemArray(void)`

Default constructor.

`~P_TempFSItemArray(void)`

Default destructor.

`bool insertFSItem(P_FSItem* fs_item)`

Inserts the fsitem into the i_{th} vector where i is the symbol ID for the attribute of the fsitem.

`bool merge(void)`

Merges the elements such that each item's attribute is it's index in the array; items with different operators will be in different columns (separate vector elements). There will be six columns in all:

for all items with operator `= (^attribute)`
for all items with operator `$ (^attribute)`
for all items with operator `=c (^attribute)`
for all items with operator `!= (^attribute)`
for all items with operator `=c (!attribute)`
for all items with operator `!= (!attribute)`

`void clear(void)`

Clears `temp_fsitem_array`.

P_FSSymbolTable

This is the symbol table for the symbols that will be used during f-description and f-structure processing.

Attributes

`static string* symbol_table`

Keeps symbols e.g., `NUM`, `GEND`, `SUBJ NUM`, etc; the ID for a symbol is (the index of the array + 1) so that no symbol is assigned the ID 0. It uses a hash function to enable 2-way access, i.e., using the ID we can directly get the string, and by using a hash function we can get the index of the string.

`static bool initialization_complete`

To indicate the end of the initializtion process after which entries in `symbol_table` will be temporary.

`static short* temp_entries`

Will have the the indices of the entries in `symbol_table` that are temporary so that they can be removed during the cleanup operation that takes place after each cycle.

`static PFSsymID GFID`

TAHIRA

static PFSsymID PredID
TAHIRA

static PFSsymID IndexID
TAHIRA

static list<PFSsymID> Subcats
TAHIRA

Functions
P_FSSymbolTable(void)
Constructor, does nothing because no object is meant to be instantiated.

~P_FSSymbolTable(void)
Destructor, does nothing because no object is meant to be instantiated.

static void init(void)
Serves as a constructor (allocates memory etc.)

static PFSsymID getSymID(string symbol)
If the symbol exists returns its ID, else it inserts the symbol and return it's ID, making the necessary updations incase the entry is meant to be temporary.

static string getSym(PFSsymID symbol)
Returns the symbol that has the indicated ID.

static PFSsymID hash(string key, int size = FSSYMBOLTABLESIZE)
Hash function that is to be used for making insertions into `symbol_table`.

static void endInitialization(void)
Indicates end of initialization and makes preparations for the temporary entries.

static void addTempIndex(int index)
When an entry is made that should be temporary it adds it's index to `temp_entries`.

static void collectSelectiveGarbage(void)
Deletes all the temporary entries from `symbol_table` and re-initializes `temp_entries`.

static void printSymbolTable(ofstream& myfile)
Prints `symbol_table`. For testing purposes only.

static void LoadSubcats(void)
TAHIRA

P_FSItem

`P_FSItem` is the abstract base class which will be used to keep the different types of fs items (attribute-value pairs) that are possible in an f-structure or an f-description.

Attributes
PFSsymID attribute
Stores the ID of the attribute (LHS) such as `GEND`, `NUM`, `SUBJ NUM` etc.

Functions

`P_FSItem(void)`
Constructor; does nothing.

`virtual ~P_FSItem(void)`
Destructor, does nothing.

`virtual P_FSItem* makeOpNotEq(void) = 0`
Changes the operator of the `P_FSItem` to `!=` (not equal). Required when creating the fs items because expressions like:

```
~[GEND =c M]
```

should convert to

```
GEND != M
```

`virtual P_FSItem* insertContext(list<contextByte> context_list) = 0`
Adds the contextlist to the `P_FSItem`.

`virtual P_FSItem* makeCopy(void) = 0`
Makes a copy of the `P_FSItem`.

`virtual bool addContextedValue(P_ContextedValue* cont_val) = 0`
Adds a contexted value to the `P_FSItem`.

`virtual int numberOfValues(void) = 0`
Returns the number of values that the `P_FSItem` has.

`virtual bool operator==(P_FSItem* right_item) = 0`
Overloaded operator to check if two `P_FSItem`s are equal.

`virtual PFSsymID getSimpleValue(void) = 0`
Gets a particular type of value from the `P_FSItem` if it has it.

`virtual byte getAttributeType(void) = 0`
Returns a byte that indicates the type of the attribute that the `P_FSItem` has.

`virtual bool containsContextedValue(P_ContextedValue* contexted_val) = 0`
Tells if the `P_FSItem` has `contexted_val` as a value.

`virtual void printItem(ofstream &myfile) = 0`
Prints the `P_FSItem`. Exists only for testing purposes.

`virtual bool eqMergeInto(P_FSItem* fs_item) = 0`
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

```
^attribute =
```

Knowing this, this function checks itself to see if it is of the same type then merges itself into `fs_item`, i.e., adds it's values to `fs_item`.

`virtual bool setMergeInto(P_FSItem* fs_item) = 0`
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

`^attribute $`

Knowing this, this function checks itself to see if it is of the same type then merges itself into `fs_item`, i.e., adds it's values to `fs_item`.

`virtual bool eqcMergeInto(P_FSItem* fs_item) = 0`
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

`^attribute =c`

Knowing this, this function checks itself to see if it is of the same type then merges itself into `fs_item`, i.e., adds it's values to `fs_item`.

`virtual bool noteqMergeInto(P_FSItem* fs_item) = 0`
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

`^attribute !=`

Knowing this, this function checks itself to see if it is of the same type then merges itself into `fs_item`, i.e., adds it's values to `fs_item`.

`virtual bool eqcDownMergeInto(P_FSItem* fs_item) = 0`
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

`!attribute =c`

Knowing this, this function checks itself to see if it is of the same type then merges itself into `fs_item`, i.e., adds it's values to `fs_item`.

`virtual bool noteqDownMergeInto(P_FSItem* fs_item) = 0`
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

`!attribute !=`

Knowing this, this function checks itself to see if it is of the same type then merges itself into `fs_item`, i.e., adds it's values to `fs_item`.

`virtual P_FSItem* instantiate(contextByte context_offset, P_FStructure* up_arrow, P_FStructure* down_arrow) = 0`
To instantiate when there is only a single option for the down arrow.

`virtual P_FSItem* instantiate(contextByte context_offset, P_FStructure* up_arrow, vector<P_ContextedValue*> down_arrows) = 0`
To instantiate when there are multiple options for the down arrow.

`bool operator<(P_FSItem* right_item)`
Overloaded operator made to help with sorting that does not actually work because it sorts the pointers. The following is used instead:

```
        struct forSort
        {
              bool operator()(P_FSItem* item1, P_FSItem* item2)
              {
                    return (item1->attribute < item2->attribute);
              }
        };
```

The call to sort is now made in the following manner:

```
        fs_items.sort(forSort());
```

where `fs_items` is a list of items. Using this the fs items can be sorted by attribute.

virtual P_FSItem* getItemForFS(void) = 0
Checks if the item is of a specific type (the type is termed 'item') and if it is returns a copy of itself.

virtual P_FSItem* getConstraintForFS(void) = 0
Checks if the item is of a specific type (the type is termed 'constraint') and if it is returns a copy of itself.

virtual P_FSItem* getLinkForFS(void) = 0
Checks if the item is of a specific type (the type is termed 'link') and if it is returns a copy of itself.

virtual list<P_ContextedValue*> getContextedValues(void) = 0
Returns its list of contexted values.

virtual list< list<contextByte> > isConsistent(void) = 0
Checks if the item is consistent within itself and sends back a list of nogoods as the result.

virtual list< list<contextByte> > isConsistentWithItem(P_FSItem* in_item) = 0
Checks if the item is consistent with `in_item` and sends back a list of nogoods as the result.

virtual P_FSItem * getSolution(list<contextByte> & solution) = 0
NAYYARA?

virtual P_FStructure* getInstConstAttrLink(void) = 0
Returns the `P_FStructure* link` from a specific type of `P_FSItem`.

virtual void makeLinksNull(void) = 0
Makes all the links null. This is required because we may need to get rid of the item itself but the links that it contains may be in use elsewhere.

virtual void removeAllValues(void) = 0
Removes all ths values from an item (meant to remove the values from a `P_FSItemComplex`, a `P_FSItemComplexInstConst` in particular).

P_FSItemSimple : P_FSItem

`P_FSItemSimple` is the simplest form that a `P_FSItem` can take. It is used to hold f-description / f-structure items such as `^GEND = F`, `^NUM = PL` etc. The only information that needs to be specified is the attribute (`GEND`) and the value (`F`). It will be assumed: 1) that there is an up-arrow (`^`) next to the attribute, 2) that the operator used is '`=`' and 3) that the context for this value is 0.

Attributes
PFSsymID value
This is the ID of the value (e.g. `F`, `PL`).

Functions

`P_FSItemSimple(void)`
Default constructor, does nothing.

`P_FSItemSimple(P_FSItemSimple* simple_item)`
Constructor, makes object identical to the one passed in the parameter.

`~P_FSItemSimple(void)`
Destructor, does nothing.

`P_FSItem* makeOpNotEq(void)`
Has to change the operator of the `P_FSItemSimple` from `=` to `!=`. Since a `P_FSItemSimple` assumes that the operator is a `=`, this function creates a new `P_FSItemComplex` such that it is identical to the `P_FSItemSimple` except that it's operator is a `!=` and returns it.

`P_FSItem* insertContext(list<contextByte> context_list)`
Used during the construction of fs items. If the context list contains only a zero, then a new identical `P_FSItemSimple` is constructed and returned. If the context list contains something other than a zero then a `P_FSItemComplex` is created that is identical to the `P_FSItemSimple`, except that it has the context list, and returned. In the first case a copy is made because when the function call is made we know that we will always get a new object and will safely be able to get rid of the old one.

`P_FSItem* makeCopy(void)`
Makes a copy of the `P_FSItemSimple` and returns it.

`bool addContextedValue(P_ContextedValue* cont_val)`
Does nothing because it is not expected to add anything; returns false to indicate that it hasn't added incase it is called; it is not expected that this will ever be called.

`int numberOfValues(void)`
Always returns 1 because it only contains one value (since it is a `P_FSItemSimple`).

`bool operator==(P_FSItem* right_item)`
Overloaded operator to check if two `P_FSItemSiimple`s are equal.

`PFSsymID getSimpleValue(void)`
Returns it's value because it is a `P_FSItemSimple` and the function needs to return a simple value.

`byte getAttributeType(void)`
Always returns a zero because a zero defines the type of attribute that is contained within a `P_FSItemSimple`. (for details see explanation for `attribute_type` in `P_FSItemComplex`)

`bool containsContextedValue(P_ContextedValue* contexted_val)`
Always returns false because a `P_FSItemSimple` does not contain a contexted value.

`void printItem(ofstream &myfile)`
Prints the item; for testing purposes only.

`bool eqMergeInto(P_FSItem* fs_item)`
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

```
^attribute =
```

Knowing this, this function will merge itself into fs_item by first creating a `P_ContextedValue` which corresponds to the value that the `P_FSItemSimple` (itself - this pointer) contains, and then adding this value to `fs_item`.

`bool setMergeInto(P_FSItem* fs_item)`
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

```
^attribute $
```

Knowing this, this function does nothing because a `P_FSItemSimple` does not correspond to the form defined above, so there will be no merging.

`bool eqcMergeInto(P_FSItem* fs_item)`
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

```
^attribute =c
```

Knowing this, this function does nothing because a `P_FSItemSimple` does not correspond to the form defined above, so there will be no merging.

`bool noteqMergeInto(P_FSItem* fs_item)`
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

```
^attribute !=
```

Knowing this, this function does nothing because a `P_FSItemSimple` does not correspond to the form defined above, so there will be no merging.

`bool eqcDownMergeInto(P_FSItem* fs_item)`
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

```
!attribute =c
```

Knowing this, this function does nothing because a `P_FSItemSimple` does not correspond to the form defined above, so there will be no merging.

`bool noteqDownMergeInto(P_FSItem* fs_item)`
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

```
!attribute !=
```

Knowing this, this function does nothing because a `P_FSItemSimple` does not correspond to the form defined above, so there will be no merging.

`P_FSItem* instantiate(contextByte context_offset, P_FStructure* up_arrow, P_FStructure* down_arrow)`
Instantiated version will be the same therefore just makes a copy and returns it.

`P_FSItem* instantiate(contextByte context_offset, P_FStructure* up_arrow, vector<P_ContextedValue*> down_arrows)`
Instantiated version will be the same therefore just makes a copy and returns it.

`P_FSItem* getItemForFS(void)`
A `P_FSItemSimple` is always of type 'item' therefore returns a copy of itself.

`P_FSItem* getConstraintForFS(void)`
A `P_FSItemSimple` is never of type 'constraint' therefore always returns NULL.

`P_FSItem* getLinkForFS(void)`
A `P_FSItemSimple` is never of type 'link' therefore always returns NULL.

`list<P_ContextedValue*> getContextedValues(void)`
Gets the value of the `P_FSItemSimple` and, makes a `P_ContextedValue` out of it, puts in in a list, and then returns the list. (note: there is only one value in `P_FSItemSimple`, but in `P_FSItemComplex` there is a list of values)

`list< list<contextByte> > isConsistent(void)`
A `P_FSItemSimple` is always consistent therefore an empty list of nogoods is returned.

`list< list<contextByte> > isConsistentWithItem(P_FSItem* in_item)`
Checks if the `P_FSItemSimple` is consistent with `in_item` and sends back a list of nogoods as the result. Because the actual consistency checking mechanism works at the value level, first a `P_ContextedValue` is required that corresponds to the value of the `P_FSItemSimple` (this). When the `P_ContextedValue` is created, all the values in `in_item` are checked with it for consistency. For each value from `in_item`, if it is a link then it is known to be inconsistent because the `P_FSItemSimple` value will never be a link, and a link cannot be consistent with a non-link, in this case the union of the context for both values is added to the list of nogoods. If the value is not a link, then a call to `isConsistent()` is made, and if it returns false then the union of the context of both values is added to the list of nogoods.

The incoming item, `in_item` will always be an fs item from the list `fs_items`, it will not be a constraint or a link. Also opposing contexts are not check for before because a `P_FSItemSimple` value always has the context 0.

`virtual P_FSItem * getSolution(list<contextByte> &solution)`
NAYYARA?

`P_FStructure* getInstConstAttrLink(void)`
A `P_FSItemSimple` can never be a constraint so returns NULL.

`void makeLinksNull(void)`
Has no links so does nothing.

`void removeAllValues(void):`
Does nothing, should never be called (cannot actually remove the value from a `P_FSItemSimple`).

P_FSItemComplex : P_FSItem

`P_FSItemComplex` is for all the complex forms that a `P_FSItem` can take.

Attributes
`byte attribute_type`
This indicates the type of f-description that is stored with the helps of bits. As described earlier, there can be several types of f-descriptions, all of these will be stored using the same structure, however as the type changes the method of storage will change and `attribute_type` will tell what type is stored.

The bit pattern is described below starting from the leftmost bit.

1 bit is reserved to indicate whether the attribute requires a plain value (`GEND`), (0) or an f-structure (`SUBJ`), (1).
1 bit is reserved to indicate whether the `LHS` (`attribute`) is simple (`GEND`), (0) or complex (`SUBJ NUM`), (1). The difference for complex expressions is that they have to be looked up and tokenized at run time before they can be processed. Since expressions of this type are very infrequent, the string tokenization will be feasible.
1 bit is reserved to indicate whether the attribute has an up-arrow (`^NUM`), (0) or down-arrow (`!NUM`), (1) beside it. (Note: This is relevant in the case of constraints only, for condition operators it is assumed that there is always an up-arrow. The actual values of these arrows when they're instantiated will not be required. Whenever there is a down-arrow, it will be resolved while unification is taking place, because any node that is being unified will have all its children complete; the value of the up-arrow is not required, it is already present.)
2 bits are reserved to indicate the type of operators (where the total number of operators is 4):

=	00
$	01
=c	10
≠	11

list<P_ContextedValue*> contexted_values
This is a list of values that the attribute can take labeled with the appropriate contexts.

static short element_no
To keep track of the number of elements that have been added to a set and to change the name of the element accordingly, i.e., ELEMENT1, ELEMENT2...

Functions
P_FSItemComplex(void)
Default constructor, does nothing.

P_FSItemComplex(P_FSItemComplex* complex_item)
Constructor, makes object identical to the one passed in the parameter.

~P_FSItemComplex(void)
Destructor, does nothing.

P_FSItem* makeOpNotEq(void)
Changes the bits (see table above) in `attribute_type` so that the operator becomes `!=`.

P_FSItem* insertContext(list<contextByte> context_list)
When this function is called we know for sure that there is only one value in the list of `P_ContextedValue`s (this is because it is called at that stage of FDVector building when items have not been merged). It adds the context list to that single value.

P_FSItem* makeCopy(void)
Makes a copy and returns it.

bool addContextedValue(P_ContextedValue* contexted_value)
Adds a value to the item.

int numberOfValues(void)
Returns the number of values that the item has.

bool operator==(P_FSItem* right_item)
Overloaded operator to check if two items are equal.

PFSsymID getSimpleValue(void)
Returns a zero to indicate that it is a `P_FSItemComplex` and is not a `P_FSItemSimple` and therefore does not contain simple value.

byte getAttributeType(void)
Returns `attribute_type`.

bool containsContextedValue(P_ContextedValue* contexted_val)
Checks if the list of values contains `contexted_val`.

void printItem(ofstream &myfile)
Prints the item, for testing purposes only.

bool eqMergeInto(P_FSItem* fs_item)
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

```
^attribute =
```

Knowing this, this function ensures that the `P_FSItemComplex` itself is of the type shown above and then it takes its values and puts them into `fs_item`.

bool setMergeInto(P_FSItem* fs_item)
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

```
^attribute $
```

Knowing this, this function ensures that the `P_FSItemComplex` itself is of the type shown above and then it takes its values and puts them into `fs_item`.

bool eqcMergeInto(P_FSItem* fs_item)
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

```
^attribute =c
```

Knowing this, this function ensures that the `P_FSItemComplex` itself is of the type shown above and then it takes its values and puts them into `fs_item`.

bool noteqMergeInto(P_FSItem* fs_item)
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

```
^attribute !=
```

Knowing this, this function ensures that the `P_FSItemComplex` itself is of the type shown above and then it takes its values and puts them into `fs_item`.

bool eqcDownMergeInto(P_FSItem* fs_item)
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

```
!attribute =c
```

Knowing this, this function ensures that the `P_FSItemComplex` itself is of the type shown above and then it takes its values and puts them into `fs_item`.

`bool noteqDownMergeInto(P_FSItem* fs_item)`
This function should only be used when it is ensured that the parameter `fs_item` contains an item of the following form:

```
!attribute !=
```

Knowing this, this function ensures that the `P_FSItemComplex` itself is of the type shown above and then it takes its values and puts them into `fs_item`.

`P_FSItem* instantiate(contextByte context_offset, P_FStructure* up_arrow, P_FStructure* down_arrow)`
Five specific cases are dealt with during instantiation:

1) when the attribute is complex (has a space, e.g., "`SUBJ NUM`") and the operator is "`=`": here instantiation should take place such that the head of the complex attribute is removed and used as the attribute for the item and a new f-structure should be created which will be it's value (i.e., a link), and this should be constructed by making a call to the recursive function `makeFStructure()`, also sending the remaining attribute string as a parameter.

2) when the attribute is complex (has a space, e.g., "`SUBJ NUM`") and the operator is "`$`": here the operator "`$`" is changed to an "`=`". This is done by first actually changing the operator that is represented in attribute byte, and then by appending the string "` SET ELEMENT`", and whatever the current element number (`element_no`) is to the attribute (which is actually an ID so we need to retrieve the string using the ID, modify the string and get a new ID). After the item has been instantiated a call is made to the function itself so that it can be instantiated (using option 1 described above).

3) when the attribute is complex (has a space, e.g., "`SUBJ NUM`") and the operator is "`=c`" or "`!=`": if the attribute for a constraint is complex it is left as it is. The values are then instantiated one by one.

4) when the attribute is non-complex (has no space, e.g., "`NUM`") and the operator is "`$`": is the same as option 2.

5) when the attribute is non-complex (has no space, e.g., "`NUM`") and the operator is "`=`", "`=c`" or "`!=`": the list of values is traversed and each is instantiated.

`P_FSItem* instantiate(contextByte context_offset, P_FStructure* up_arrow, vector<P_ContextedValue*> down_arrows)`
This is exactly the same as the `instantiate()` above except that when he values are instantiated the call is made to an overloaded function that can handle multiple values for the down-arrow.

`P_FSItem* getItemForFS(void)`
If the `P_FSItemComplex` itself is not a constraint, and it's not an link, returns a copy else returns NULL.

`P_FSItem* getConstraintForFS(void)`
Checks if the operator is a constraint ("`=c`" or "`!=`"), if it is makes a copy and returns it, else returns NULL.

`P_FSItem* getLinkForFS(void)`
Checks if the operator is a link ("`^=!`"), if it is makes a copy and returns it, else returns NULL.

`list<P_ContextedValue*> getContextedValues(void)`
Returns `contexted_values`.

`void makeFStructure(P_FStructure* fs, string attr, P_ContextedValue* in_value)`

This function is used during the instantiation of items that have complex attributes, i.e., attribute names that are separated by spaces, e.g., ”SUBJ NUM”, ”XCOMP SUBJ PRONTYPE” etc. The parameters inlcude fs, the f-structure into which the value has to be inserted, attr, the attribute in string format (can be complex or non-complex, i.e., with or without spaces) and in_value, the value that the attribute will eventually have assigned to it, e.g., for ”SUBJ NUM” the value could be SG, and will be the eventual value when the breakdown is complete and the attribute is NUM.

Depending on whether attr still contains spaces or not, two things can happen. If there is a space in attr, then an fs item is constructed whose value is the head of attr (e.g. SUBJ from ”SUBJ PRONTYPE”) and it's value is an empty f-structure. This empty f-structure and the remaining string, along with the value are sent as parameters to recursive call to the same function. This goes on until attr loses all it's spaces, then an fs item is made where the attribute is the ID for attr and it's value is in_value (this is the second thing from the two menioned earlier, when there are no spaces in attr).

list< list<contextByte> > isConsistent(void)
All values are checked with each other for consistency and a list of nogoods is returned as the result.

list< list<contextByte> > isConsistentWithItem(P_FSItem* in_item)
Checks if the item itself and in_item are consistent, to do this each value of in_item is checked for consistency against each value of this item. The incoming item, in_item, here is always an item from fs_items (i.e. not a ^=!); the attributes have already been matched so only values need to be checked.

virtual P_FSItem * getSolution(list<contextByte> & solution)
NAYYARA?

P_FStructure* getInstConstAttrLink(void)
Its purpose is to get a link when it is in an instantiated version of a P_FSItem, i.e., P_FSItemComplexInstConst which is derived from P_FSItemComplex.

virtual P_FStructure* getAttrLink(void)
Is meant to get the link mentioned above but since P_FSItemComplex does not have it, the class that is derived from it does, this function returns a NULL.

void makeLinksNull(void)
Goes through its list of values and makes any links that they have NULL.

void resetElementNo(void)
This is used to set the variable element_no back to 1 after each cycle so that the numbering of elements in a set can start from 1 in the new cycle.

void removeAllValues(void)
Empties the list contexted_values.

P_FSItemComplexInstConst : P_FSItemComplex

This class exists for instantiated versions of P_FSItems that are constraints (=c or !=).

Attributes

P_FStructure* link
Is the P_FStructure that the up-arrow or down-arrow is supposed to point to after instantiation.

Functions

P_FSItemComplexInstConst(void)
Default constructor.

~P_FSItemComplexInstConst(void)
Default destructor.

P_FSItemComplexInstConst(P_FSItem* fs_item, P_FStructure* up_arrow, P_FStructure* down_arrow)
When a constraint is uninstantiated it is in a `P_FSItemComplex`. This constructor takes the uninstantiated constraint and makes an instantiated constraint by inserting the `P_FStructure` that is the link.

P_FStructure* getAttrLink(void)
Returns `link`.

P_ContextedValue
This is the base class that is used for the list of contexted values for each attribute.

Attributes
list<contextByte> context
Different contexts will be represented by numbers; to denote two opposite contexts, a number and its negative value will be used, e.g., b and ⌐b can be represented by 1 and -1. Each element in the list will be a conjunct, e.g., for the expression a & b & c, each of a, b and c will be in a list element.

Functions
P_ContextedValue(void)
Constructor, does nothing.

virtual ~P_ContextedValue(void)
Destructor, does nothing.

virtual bool operator==(P_ContextedValue* right_value) = 0
Overloaded operator to see if two values are equal.

virtual PFSsymID getPlainVal(void) = 0
To return a specific type of value, there are four types that will be inherited from `P_ContextedValue` (all four are described in the four following sections), in the virtual function for the correct type the ID for the value will be returned, the rest will return 0 to indicate that there is no value of the type required i.e. a plain value.

virtual PFSsymID getFDLinkVal(void) = 0
To return a specific type of value, there are four types that will be inherited from `P_ContextedValue` (all four are described in the four following sections), in the virtual function for the correct type the ID for the value will be returned, the rest will return 0 to indicate that there is no value of the type required i.e. an f-description with link (FDLink) value.

virtual byte getValueType(void) = 0
Returns value_type (byte describing the type of the value, does not exist in all four value types, see the following four sections for details).

virtual int getNoInPlainValList(void) = 0
One class derived from `P_ContextedValue` actually has a list of values. This function is required to get the number of values in that list.

virtual bool containsVal(PFSsymID value) = 0
Only used when the value is a value list , e.g., `{SG, PL}`, checks if the value in the parameter matches one of the list values.

virtual void printValue(ofstream& myfile) = 0
Prints all the values, for testing purposes only.

virtual P_ContextedValue* instantiate(P_FStructure* up_arrow, P_FStructure* down_arrow, contextByte offset) = 0
Instantiates the values.

virtual list<P_ContextedValue*> instantiate(P_FStructure* up_arrow, vector<P_ContextedValue*> down_arrows, contextByte offset) = 0
Instantiates the values when there are multiple options for the down-arrow.

virtual bool zeroValue(void) = 0
To check if the values side is zero (i.e., has no value, e.g., in cases like `^=!`).

void setContext(contextByte offset)
Sets the context according to the offset sent as the parameter.

virtual void setFSLinkValue(PFSsymID value) = 0
Sets `value` for fs link type objects.

virtual void setFSLinkValueType(byte value_type) = 0
Sets `value_type` for fs link type objects.

void addContexts(list<contextByte> add_context)
Adds `add_context` to the value `context`.

virtual P_FStructure* getFSLinkLink(void) = 0
Gets `link` from a link type object.

virtual bool isConsistent(P_ContextedValue* cont_value) = 0
Checks if the `P_ContextedValue` is consistent with `cont_value`.

virtual list<PFSsymID> getPlainValList(void) = 0
Returns `value` if the `P_ContextedValue` is a plain value, returns 0 otherwise.

virtual PFSsymID getFSLinkValue(void) = 0
Gets `value` for fs link type objects.

virtual P_FStructure* getFSLink(void) = 0
TAHIRA

virtual P_ContextedValue* makeCopy(void) = 0
Returns a copy of the `P_ContextedValue`.

virtual void makeLinksNull(void) = 0
Makes all the links NULL, this is required because when copies are made, or new versions are made they will be pointing to the same links and we don't want the links to disappear when we get rid of the originals.

P_ConValPlainVal : P_ContextedValue

Used to store simple values like `SG`, `PL` etc. (Is not the same thing as `P_FSItemSimple` because these values will have contexts attached to them.)

Attributes
PFSsymID value
Is the ID of the value (e.g. ID for the value "`PL`").

Functions

`P_ConValPlainVal(void)`
Constructor, does nothing.

`~P_ConValPlainVal(void)`
Destructor, does nothing.

`bool operator==(P_ContextedValue* right_value)`
Returns true if `value` and `context` match, else returns false.

`PFSsymID getPlainVal(void)`
Returns `value`.

`PFSsymID getFDLinkVal(void)`
Is not an FDLink type value therefore returns zero.

`byte getValueType(void)`
Returns a bit pattern that indicates that it has no `value_type`.
(`NOVALTYPE = 63 = 00111111`, see description of `value_type` for details.)

`int getNoInPlainValList(void)`
Returns zero because it is not a PlainValList type `P_ContextedValue`.

`bool containsVal(PFSsymID value)`
Returns zero because this function is meant to be called only for `P_ConValPlainValList` type objects.

`void printValue(ofstream& myfile)`
Prints the value, for testing purposes only.

`P_ContextedValue* instantiate(P_FStructure* up_arrow, P_FStructure* down_arrow, contextByte offset)`
Makes a copy of itself, sets it's context using `offset` and returns it (in a `P_ConValPlainVal` there is no up-arrow or down-arrow that needs to be set during instantiation).

`bool zeroValue(void)`
Should always return false because an "`^=!`" type item is being searched for when this is called.

`list<P_ContextedValue*> instantiate(P_FStructure* up_arrow, vector<P_ContextedValue*> down_arrows, contextByte offset)`
Is no different from the `instantiate()` described before, except that the instantiated value is put in a list and returned. This is because this overloaded version is for cases when there are multiple options for the down-arrows, but since there are no down-arrows in a `P_ConValPlainVal`, nothing needs to be done here.

`void setFSLinkValue(PFSsymID value)`
Does nothing, should never be called because it is a `P_ConValPlainVal`, not a `P_ContValFSLink`.

`void setFSLinkValueType(byte value_type)`
Does nothing, should never be called because it is a `P_ConValPlainVal`, not a `P_ContValFSLink`.

`P_FStructure* getFSLinkLink(void)`
Has no FSLink value so returns NULL.

`bool isConsistent(P_ContextedValue* cont_value)`
Checks for consistency. This type of object can only be consistent if the `cont_value` in the parameter is either a `P_ConValPlainVal` or a `P_ConValPlainValList`. If `cont_value` is a `P_ConValPlainVal` and `value` for

both of them are the same, then they're consistent. If `cont_value` is a `P_ConValPlainValList`, and if any one of the values in the list is equal to `value` then it is consistent. Examples:

`SG` & `{SG, PL}`	consistent
`SG` & `SG`	consistent
`SG` & `PL`	not consistent
`PL` & `{X, Y}`	not consistent

`list<PFSsymID> getPlainValList(void)`
Returns an empty list because it has no list.

`PFSsymID getFSLinkValue(void)`
Returns -1 because it has no FSLinkValue, a zero is not returned because that is a valid value for an FSLinkValue. Here we need to indicate that this is a type that does not have an FS link.

`P_FStructure* getFSLink(void){return 0;}`
TAHIRA

`P_ContextedValue* makeCopy(void)`
Makes a copy and returns it.

`void makeLinksNull(void)`
Has no links so does nothing.

P_ConValPlainValList : P_ContextedValue

Used to store a list of simple values, e.g., `{SG, PL}`.

Attributes
`list<PFSsymID> values`
List of IDs representing the values in the list.

Functions
`P_ConValPlainValList(void)`
Constructor, does nothing.

`~P_ConValPlainValList(void)`
Destructor, does nothing.

`bool operator==(P_ContextedValue* right_value)`
Checks three things: 1) that the context is the same, 2) that the number of values in both value lists is the same and 3) that the list in `right_value` contains all the elements that the list of the object itself contains.

`PFSsymID getPlainVal(void)`
Is not a plain value so returns 0.

`PFSsymID getFDLinkVal(void)`
Has no FDLink values so returns 0.

`byte getValueType(void)`
Returns a bit pattern that indicates that it has no `value_type`.
(`NOVALTYPE = 63 = 00111111`, see description of `value_type` for details.)

`int getNoInPlainValList(void)`
Returns the number of values in the `values`.

`bool containsVal(PFSsymID value)`
Checks if `value` is one of the elements of the list `values`.

`void printValue(ofstream& myfile)`
Prints the value, for testing purposes only.

`P_ContextedValue* instantiate(P_FStructure* up_arrow, P_FStructure* down_arrow, contextByte offset)`
Makes a copy of itself, sets it's context using `offset` and returns it (in a `P_ConValPlainValList` there is no up-arrow or down-arrow that needs to be set during instantiation).

`bool zeroValue(void)`
Should always return false because an "`^=!`" type item is being searched for when this is called.

`list<P_ContextedValue*> instantiate(P_FStructure* up_arrow, vector<P_ContextedValue*> down_arrows, contextByte offset)`
Is no different from the `instantiate()` described before, except that the instantiated value is put in a list and returned. This is because this overloaded version is for cases when there are multiple options for the down-arrows, but since there are no down-arrows in a `P_ConValPlainValList`, nothing needs to be done here.

`void setFSLinkValue(PFSsymID value)`
Does nothing, should never be called because it is a `P_ConValPlainValList`, not a `P_ContValFSLink`.

`void setFSLinkValueType(byte value_type)`
Does nothing, should never be called because it is a `P_ConValPlainValList`, not a `P_ContValFSLink`.

`P_FStructure* getFSLinkLink(void)`
Has no FSLink value so returns NULL.

`bool isConsistent(P_ContextedValue* cont_value)`
If the incoming `cont_value` is a `P_ConValPlainVal` and one of the element is values matches it's value then returns true, otherwise false. If the incoming `cont_val` is a `P_ConValPlainValList` and any one of thier values match then returns true, otherwise false. Examples:

`A` & `{A, B}`	consistent
`{A, B}` & `{B, C}`	consistent
`A` & `{B, C}`	not consistent
`{A, B}` & `{C, D}`	not consistent

`list<PFSsymID> getPlainValList(void)`
Returns `values`.

`PFSsymID getFSLinkValue(void)`
Returns -1 because it has no FSLinkValue, a zero is not returned because that is a valid value for an FSLinkValue. Here we need to indicate that this is a type that does not have an FS link.

`P_FStructure* getFSLink(void){return 0;}`
TAHIRA

`P_ContextedValue* makeCopy(void)`
Makes a copy and returns it.

`void makeLinksNull(void)`
Has no links so does nothing.

P_ConValFDLink : P_ContextedValue

`P_ConValFDLink` will be used to keep only f-descriptions and not f-structures and therefore will always be uninstantiated. When this type is instantiated it will be converted into `P_ContValFSLink`.

Attributes

`PFSsymID value`

Is the ID assigned to the value (`SUBJ`, `SUBJ NUM`).

`byte value_type`

Uses bits to show what type of value is stored. The bit pattern is described below starting from the leftmost bit.

> **1 bit** is reserved to tell whether there is an `^` (0) or a `!` (1) next to the value.
> **1 bit** is reserved to tell if a complex expression is stored (`SUBJ NUM`), the difference for complex expressions like these is that they have to be looked up and tokenized at run time before they can be processed. Since expressions of this type are very infrequent, the string tokenization will be feasible.

Functions

`P_ConValFDLink(void)`

Constructor, does nothing.

`~P_ConValFDLink(void)`

Destructor, does nothing.

`bool operator==(P_ContextedValue* right_value)`

Returns true if `context`, `value` and `value_type` match.

`PFSsymID getPlainVal(void)`

Returns zero because it does not have a plain value.

`PFSsymID getFDLinkVal(void)`

Returns `value` (because this is a `P_ConValFDLink`).

`byte getValueType(void)`

Returns `value_type`.

`int getNoInPlainValList(void)`

Returns zero because it does not have a list.

`bool containsVal(PFSsymID value)`

Returns zero because this function is meant to be called only for `P_ConValPlainValList` type objects.

`void printValue(ofstream& myfile)`

Prints the value, for testing purposes only.

`P_ContextedValue* instantiate(P_FStructure* up_arrow, P_FStructure* down_arrow, contextByte offset)`

Makes a new `P_ContValFSLink` type object (because this is the tye that the `P_ConValFDLink` will become when it is instantiated) that has the same `context`, `value` and `value_type` as the `P_ConValFDLink`. After this using `value_type`, the type of arrow next to the value (up-arrow or down-arrow) is determined, if it is found to be an up-arrow, link (in `P_ContValFSLink`) is pointed to `up_arrow`, if it is a down-arrow, then to `down_arrow`. After this `context` is adjusted according to `offset`.

`bool zeroValue(void)`

Returns true if `value` is zero, false otherwise.

`list<P_ContextedValue*> instantiate(P_FStructure* up_arrow, vector<P_ContextedValue*> down_arrows, contextByte offset)`
The essence here is the same as the `instantiate()` above except that instead of the `down_arrow P_FStructure*` we get a set a `P_ContextedValue`s which have their down-arrows already pointing to the required postions. So if `value_type` shows that we have a down-arrow next to the value, this function, for each down-arrow value, makes a copy of it and then sets the copy's `value`, `value_type` and `context` such that it is the same as it's own. This results in a list of instantiated values.

In case there is an up-arrow next to the value the function proceeds like the earlier version, except that the instantiated value is inserted into a list.

In both cases `context` is adjusted using `offset`.

`void setFSLinkValue(PFSsymID value)`
Does nothing, is not expected to be called (is not an FSLink value).

`void setFSLinkValueType(byte value_type)`
Does nothing, is not expected to be called (is not an FSLink value).

`P_FStructure* getFSLinkLink(void)`
Returns NULL because it does not have an FS link.

`bool isConsistent(P_ContextedValue* cont_value)`
Should never be called because only instantiated values need to be checked for consistency, but returns true to be on the safe side.

`list<PFSsymID> getPlainValList(void)`
Returns an empty list (has no list to return).

`PFSsymID getFSLinkValue(void)`
Returns -1 because it has no FSLinkValue, a zero is not returned because that is a valid value for an FSLinkValue. Here we need to indicate that this is a type that does not have an FS link.

`P_FStructure* getFSLink(void){return 0;}`
TAHIRA

`P_ContextedValue* makeCopy(void)`
Makes a copy and returns it.

`void makeLinksNull(void)`
Has no links so does nothing.

P_ContValFSLink : P_ContextedValue

`P_ContValFSLink` will be used to store items of f-structures only (not f-descriptions) and will therefore always be in instantiated form. Values like `!NUM` and `!SUBJ NUM` will be stored in this.

Attributes

`PFSsymID value`
Is the ID assigned to the value (`SUBJ`, `SUBJ NUM`).

`P_FStructure* link`
Is the f-structure that the up-arrow or down-arrow next to the value is pointing to. The direction of the arrow is not signifcant at this point but it can be obtained through `value_type`.

`byte value_type`

Uses bits to show what type of value is stored. The bit pattern is described below starting from the leftmost bit.

> **1 bit** is reserved to tell whether there is an `^` (0) or a `!` (1) next to the value.
> **1 bit** is reserved to tell if a complex expression is stored (`SUBJ NUM`), the difference for complex expressions like these is that they have to be looked up and tokenized at run time before they can be processed. Since expressions of this type are very infrequent, the string tokenization will be feasible.

Functions

`P_ContValFSLink(void)`
Default constructor, initializes `link` to NULL.

`P_ContValFSLink(P_FStructure *link, PFSsymID value)`
Constructor, sets `link` and `value` to those sent as parameters.

`~P_ContValFSLink(void)`
Destructor, deletes `link` (if it is not NULL), and then sets it to NULL. This is done to ensure that single `P_FStructure`s that are pointed to by multiple links are only deleted once.

`bool operator==(P_ContextedValue* right_value)`
Always returns false because no items of this type are instantiated when this operator is used.

`PFSsymID getPlainVal(void)`
Has no plain value, returns zero.

`PFSsymID getFDLinkVal(void)`
Has no FDLink value, returns zero.

`byte getValueType(void)`
Returns `value_type`.

`int getNoInPlainValList(void)`
Returns zero because it has no list.

`bool containsVal(PFSsymID value)`
Returns false because this function is meant to check if the value list contains a specific element, but there is no list here.

`void printValue(ofstream& myfile)`
Intended for testing purposes but not implemented because it isn't needed yet.

`P_ContextedValue* instantiate(P_FStructure* up_arrow, P_FStructure* down_arrow, contextByte offset)`
Returns NULL, it should never be called because a `P_ContValFSLink` is already instantiated.

`bool zeroValue(void)`
Returns true if `value` is zero, false otherwise.

`list<P_ContextedValue*> instantiate(P_FStructure* up_arrow, vector<P_ContextedValue*> down_arrows, contextByte offset)`
Returns an empty list, it should never be called because a `P_ContValFSLink` is already instantiated.

`void setFSLinkValue(PFSsymID value)`
Sets `value`.

void setFSLinkValueType(byte value_type)
Sets `value_type`.

P_FStructure* getFSLinkLink(void)
Returns `link`.

bool isConsistent(P_ContextedValue* cont_value)
Returns true, but should never be called. All consistency checking for values only takes place with `P_ConValPlainVal`s and `P_ConValPlainValList`s. This works because to check a link for consistency we have to check it's values, and in essence the actual concrete values are `P_ConValPlainVal`s and `P_ConValPlainValList`s.

list<PFSsymID> getPlainValList(void)
Has no list so returns an empty list.

PFSsymID getFSLinkValue(void)
Returns `value`.

P_FStructure* getFSLink(void){return link;}
TAHIRA

P_ContextedValue* makeCopy(void)
Makes a copy and returns it.

void makeLinksNull(void)
Sets the value of `link` to NULL.

P_FStructure

Represents a single f-structure, where `P_FSItem`s are used to represent attribute-value pairs. If the value of an attribute is an f-structure itself, it will be stored in another `P_FStructure`, and in the place of the value, a pointer to that f-structure will be given. An example of this representation is shown, where the following f-structure:

```
┌          ┌              ┐ ┐
| SUBJ     |NUM         SG| |
|          |GEND M        | |
|          └              ┘ |
| ABC       abc             |
└                           ┘
```

is represented below:

0	1		
SUBJ 1	NUM SG		
ABC abc	GEND M		

Attributes

list<P_FSItem*> fs_items
A list to keep all fs items that are 1) NOT constraints, and 2) NOT `^=!` type fs items.

list<P_FSItem*> constraints
A list to keep all the fs items that are constraints, e.g., `^NUM =c SG`, `^NUM != M` etc.

P_FSItem* links
A `P_FSItem` where all fs items of type `^=!` are kept (multiple values fot the down-arrow will be kept when required).

list< list<contextByte> > nogoods
List of nogood expressions.

list<contextByte> first_solution
NAYYARA

bool conts_checked
TAHIRA

bool cc_checked
TAHIRA

bool failed
NAYYARA

bool nogoods_extracted
NAYYARA

Functions

P_FStructure(void)
Default constructor, initializes `links` by pointing to it to a `P_FSItemComplex` that has the attribute `^=` and no values, as a result links can be used to keep fs items of the form `^=!`. TAHIRA.

~P_FStructure(void)
Default destructor, deletes `links` and all the items in `fs_items` and `constraints`.

void addItem(P_FSItem* fs_item)
The main idea behind adding an item to an f-structure is to add them such that an attribute occurs only once in the f-structure. This means that when adding an fs item, if there isn't already an item in the f-structure with the same attribute then the whole item can be added in. If an item with the same attribute already exists then the values from the incoming item need to be removed and added to the previous item. In this way, items will be constructed that will have multiple values.

Also, this function is only called when the incoming item, `fs_item`, is definitely an fs item, not a link and not a constraint.

If an attribute is such that it requires a link on the value side then that attribute should have a link as it's value, and the f-structure that is pointed to by that link will contain the actual link values in the links variable; this implies that items such as "`^SUBJ = !SUBJ`" will get stored such that the value link will be in the links of the newly created f-structure but the value part ("`SUBJ`") of "`!SUBJ`" will not be placed anywhere explicitly.
The following are examples that qualify for this special treatment:
1) `^SUBJ = !SUBJ`
2) `^SUBJ NUM = SG` (which will have been transformed to "`^SUBJ = !`" at this point)
(if there is a long chain like the one above then only the head will be treated this way, the rest of it will remain as originally built in instantiate)

The following are examples that DO NOT qualify for this special treatment:
1) `^NUM = !NUM`
2) `^NUM = SG`

void addConstraint(P_FSItem* constraint)
This is only called when the `P_FSItem constraint` is definitely a constraint, not an item or a link; if an item with the same attribute type exists, adds only the new values to that, else puts the whole item in; constraints can be of two types: 1) `^attribute =c` and 2) `^attribute !=`. Constraints that have the same attribute and the same

operator will be put into one item. So if for example we get a "`^NUM =c SG`" to add and we already have a "`^NUM =c PL`", then we add the value `SG` to the item that is already present. If there is no "`^NUM =c`" type constraint present that the whole fs item is added.

`void addLink(P_FSItem* link)`
This is only called when the `P_FSItem` sent as parameter is definitely a link ("`^=!`" only), not an item or a constraint; if an item with the same attribute type exists, adds only the new values to that, else puts the whole item in; here since the attribute is always the same (i.e. nothing), the value will always be added to the item.

`void unify(void)`
3 things are to be done here:
1. check if each fsitem from the list `fs_items` is consistent within itself;
2. check if each item in the list `fs_items` is consistent with each link in the list `links`;
3. check if each link in list `links` is consistent with all the other links in the list.
NAYYARA

`void addNogood(list<contextByte> nogood)`
Adds a nogood. TAHIRA

`void CheckCompleteness()`
TAHIRA

`list<P_FSItem*> SearchInItemsLinks(PFSsymID Att)`
TAHIRA (`search Att(SUB OBJ NUM etc) in This FS`)

`bool SearchItemsDeepWithContext(PFSsymID att,list<P_FSItem*> &ResultList, list<list<contextByte> > &exContList, list<contextByte> extCont)`
TAHIRA

`P_FSItem* SearchInItemsUnordered(PFSsymID att)`
TAHIRA (`does not search in links and constraints, does not assume any order of fs-items`)

`P_FSItem* SearchInConstraintsUnordered(PFSsymID att)`
TAHIRA

`list<P_FSItem*> GetItemsDeep()`
TAHIRA

`list<P_ContextedValue*> FindValuesDeep(PFSsymID att, list<list<contextByte> > &nogoodsbelow)`
TAHIRA (`can find values with nogoods, deep down in links (only one level of depth is assumed)`)

`list<P_ContextedValue*> FindComplextValuesDeep(string long_att, list<list<contextByte> > &nogoodsbelow,list<contextByte> & ext_context)`
TAHIRA (`can find complex values deep down in links (only one level of depth is assumed)`)

`void resolveValueLinks(void)`
TAHIRA (`puts actual values in the place of !featurename like !NUM`)

`list<list<contextByte> > SearchSubcatsDeepWithContext(vector<list<P_ContextedValue*> > &ResultList, vector<list<list<contextByte> > > &exContList)`
TAHIRA

list<list<contextByte> > SearchSubcatsDeepWithContext2(vector<list<P_ContextedValue*> > &ResultList)
TAHIRA

bool inNogoods(list<contextByte> context)
TAHIRA

P_FStructure* extractSolutionDeep(list<contextByte> & ext_solution)
TAHIRA

list<list<contextByte> > GetNogoodsDeep()
TAHIRA

list<P_ContextedValue*> FindValuesDeepWithContext(PFSsymID att, list<list<contextByte> > &nogoods_below, list<contextByte> & ext_context)
TAHIRA

list< list<contextByte> > isConsistentWithItem(P_FSItem* in_item)
Checks itself for consistence with `in_item` and sends back a list of nogoods as the result. To do his it does the following:
1. check the fs_item with the element of list `fs_items` that matches it's attribute
2. checks the fs_item with all the f-structures in `links`

void checkItemsConsistency(void)
Iterates through all the items in `fs_items` and checks if they're consistent. It also checks which of the items have a dummy f-structure (see previous sections for details on the dummy f-structure), and if they do it calls `unify()` for them because the current path is the only one through which they can be traced and unified.

void checkItemsLinksConsistency(void)
Each item in `fs_items` is checked with all the f-structures in `links`.

void checkLinkLinksConsistency(void)
Checks all the f-structures in `links` for consistency with each other. Here, as a result of inconsistency, nogood expressions are added to `nogoods`, but before adding each nogood, the union of contexts of the two links (that were being checked for consistency) is added to the nogood.

list< list<contextByte> > isConsistentWithFS(P_FStructure* in_fs)
All items and links in one f-structure are to be checked for consistency with all the items and links in the other f-structure.

void checkConstraints(P_FStructure *FS, list<contextByte> ext_context)
For each constraint, get the attribute link, e.g., in ”`^NUM =c SG`”, the attribute link will be the f-structure pointed to by the up-arrow next to ”`NUM`”. For this f-structure, call `checkConstraint()`, sending `constraint` as the parameter. Also, this function itself is called for all the values in `links` and all the `fsitems` that have links as values.

list< list<contextByte> > checkConstraint(P_FSItem* constraint,list<bool*> &FoundFlags)
3 things are done here:
1) check the constraint with all the items (different from consistency checking because
    a. if attribute not there then fail;
    b. attributes like "`SUBJ NUM`" have not been resolved;)

2) check the constraint with all the links (need a function here that gets the constraint as a parameter and checks if it's ok with the f-structure) - also call `checkConstraints()` for all the links;

3) for all the pure links in the items list (like ”`^SUBJ = !`”) - those which are stored within a dummy f-structure - call the consistency check for all the down-arrows (these will have the `FSATTR` bit set).

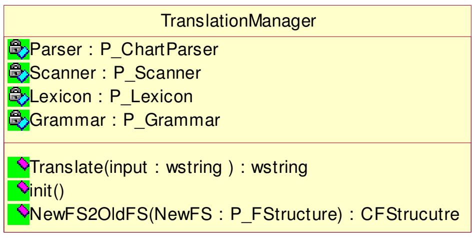
`P_FStructure* extractSolution( list<contextByte> & ext_solution)`
NAYYARA

`list<P_ContextedValue*> getComplexValues(string string_trace)`
`string_trace` is a string that represents an attribute with the head removed, e.g., ”`NUM`”, which is the result when the head (”`SUBJ`”) is removed from ”`SUBJ NUM`”, or ”`Y Z`” from ”`X Y Z`”. This f-structure (which calls this function) is the dummy f-structure that is the value of the head, and it will have the relevant links inserted in it. This function is a recursive function that traces out the required `P_ContextedValue`s and returns them, e.g., for ”`Y Z`” it will retrieve the values for `Z`, and in the case of ”`NUM`” it will retrieve it’s values.

`list< list<contextByte> > getNogoods(void)`
Returns `nogoods`.

`list<P_FSItem*> getConstraintsFromLinks(void)`
Is a recursive function that collects all the constraints of this f-structure, and all the constraints that it finds in f-structures that it’s `links` point to, and all the f-structures that their links point to and so on.

TranslationManager

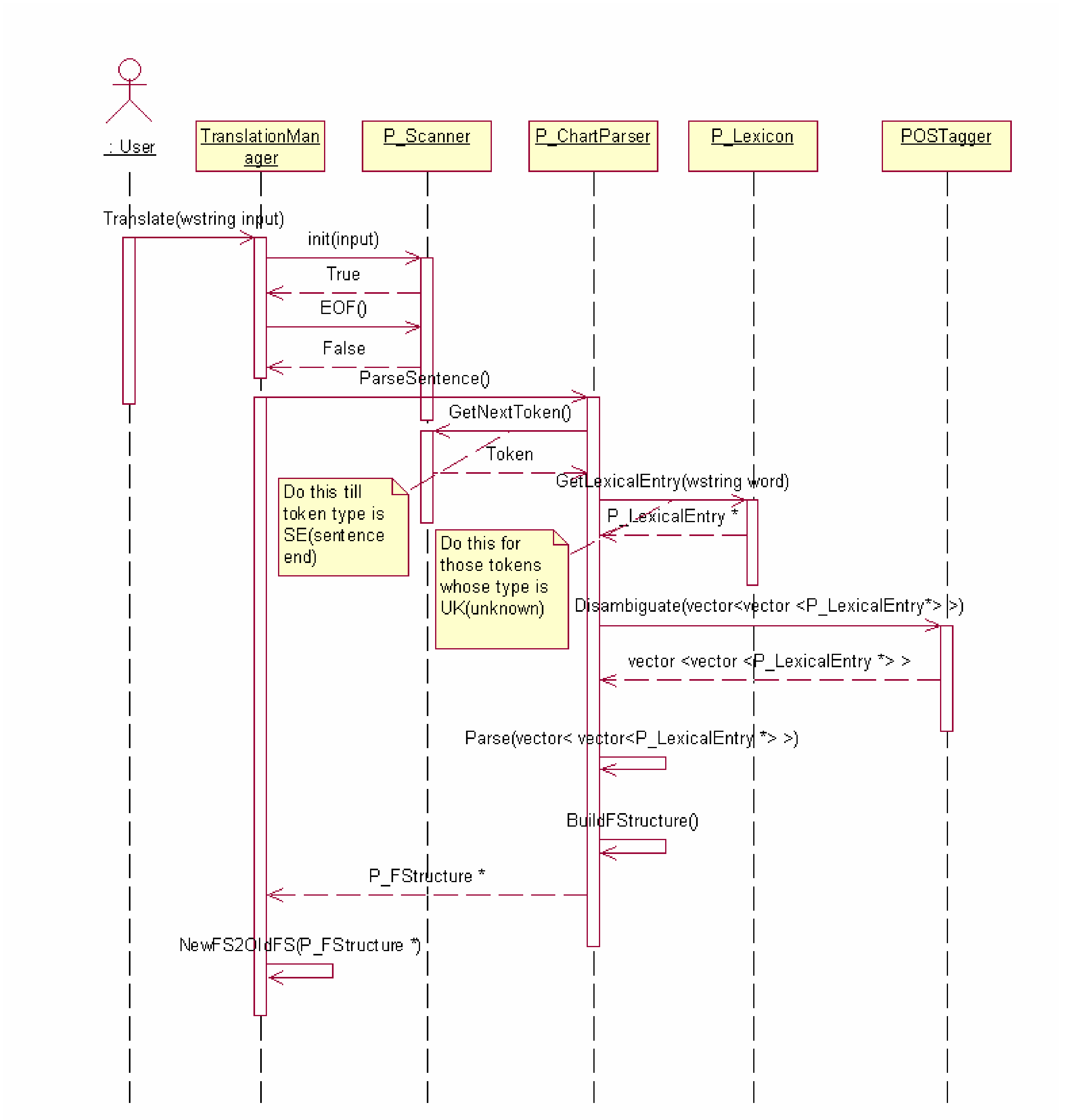


2.2. Sequence Diagrams

2.2.1. Initialization

2.2.2. Parsing

Primary Scenario



: User
TranslationManager
P_Scanner
P_ChartParser
P_Lexicon
POSTagger
Translate(wstring input)
init(input)
True
EOF()
False
ParseSentence()
GetNextToken()
Token
GetLexicalEntry(wstring word)
P_LexicalEntry *
Do this till token type is SE(sentence end)
Do this for those tokens whose type is UK(unknown)
Disambiguate(vector<vector <P_LexicalEntry*> >)
vector <vector <P_LexicalEntry *> >
Parse(vector< vector<P_LexicalEntry *> >)
BuildFStructure()
P_FStructure *
NewFS2OldFS(P_FStructure *)

2.3. Algorithms

```
P_ChartParser:: P_ChartParser(P_Lexicon * lexptr, P_Grammar * gramptr,
                              P_Scanner * scanptr, long Time, long Space)

LexiconPtr := lexptr
ScannerPtr := scanptr
GrammarPtr := gramptr
TimeLimit := Time
SpaceLimit := Space

End Function

P_FStructure * P_ChartParser:: Parse()

StartTime : = now
SpaceCount := 0

While Token POS is not se
      Get Token from scanner
      If Token has some POS
            Get its Lex Entry from LexPtr
      Else (POS is unknown)
            Get Lex entry for lexeme from LexPtr
            If Lex Entry not found
                  Give it POSes and Dummy Lex entries of adj, adv, n, v

Chart.resize(sentense-length+1)
//disambiguate POSes
//convert all lex entries to lex items (of type P_ChartItem with BackPointers
containing Null)
//Give every chart Column its lex items
//call ParseSentence

End Function

P_FStructure * P_ChartParser:: ParseSentence()

CurrentTime : long
CurrentColumn := 0
int FirstSet := 0

For each lexical entry in the current chart column
      FirstSet |= (POS of the lexical entry)

vector<P_Production *> prod_vector
prod_vector := GrammarPtr.GetProduction(GrammarPtr.GetStartSymbolId(),FirstSet)

for each prodptr in prod_vector
      ChartItem citem(0,prodptr)
      Chart[CurrentColumn].LoadItem(citem)

P_Symbol sym

While(CurrentColumn<=ChartSize)

     While Chart[CurrentColumn].GetCurrentIndex()<Chart[CurrentColumn].GetItemCount()

            citemptr := Chart[CurrentColumn].GetCurrentItem()

            if citemptr->IsComplete()
```

```
                  Complete()
            Else

                  sym := citemptr->GetCurrentSymbol()
                  if sym.IsTerminal
                        Scan()
                  Else
                        Predict()
                  End if

            End if

            Chart[CurrentColumn].MoveNext();

            CurrentTime = now

            If CurrentTime-StartTime>TimeLimit

                  Release chart memory
                  Return Null

            If SpaceCount>SpaceLimit

                  Release chart memory
                  Return Null

      End While

End While
```

End Function

void P_ChartParser:: Predict()

```
P_Symbol sym
sym:=Chart[CurrentColumn].GetCurrentItem()->GetCurrentSymbol()

For each lexical entry in the current chart column
      FirstSet |= (POS of the lexical entry)

vector<P_Production *> prod_vector
prod_vector := GrammarPtr.GetProduction(sym,FirstSet)

for each prodptr in prod_vector
      ChartItem citem(0,prodptr)
      LoadInChart(citem, CurrentColumn)
```

End Function

void P_ChartParser:: Scan()

```
P_Symbol sym := current symbol of current item of current chart column

For each lexical entry item in the current chart column
      If POS of lexical entry == sym
            citem := current item of current chart column
            citem.MoveDot(pointer to lexical entry item)
            LoadInChart(citem, CurrentColumn+1)
            break
```

End Function

void P_ChartParser:: Complete()

```
Chart[CurrentColumn].GetCurrentItem->ComputeProbability()

int StartColIndex = start index of current item of current chart column
P_Symbol sym := LHS of production of current item of current chart column

vector<byte> *indices := Chart[StartColIndex].GetProdsWithDotAt(sym)

for each index in indices
      citem := Chart[StartColIndex].GetItemAt(index)
      citem.MoveDot(pointer of current item of current chart column)
      LoadInChart(citem, CurrentColumn)


End Function

bool P_ChartParser::LoadInChart(P_ChartItem &item, int col)

return Chart[col].LoadItem(item)

End Function
```

```
bool P_ChartColumn::LoadItem(P_ChartItem &item)

vector<int> indices:=ProductionIndex.find(prod id of item's production)

for each index in indices
      if items[index].GetDotPos()==item.GetDotPos()
            items[index].AddBackPtr(item.GetBackPtrs())
            return false


items.pushback(item)

ProductionIndex[item.GetProdPtr().LHS].pushback(ItemCount)
ProdsWithDotAt[item.GetCurrentSymbol()].pushback(ItemCount)
ItemCount++

return true

End Function
```

```
void CGrammar::lfg2cfg()

//input: a list of LFG productions
//output: a list of CFG productions

{
      stack<symbol> stk1
      stack<symbol> stk2

      for(k=0;k<vlist.size();k++)
      {
            prod=vlist[k]
            len=prod.RHS.size()

            for(p=0;p<len;p++)
```

```
                {
                        symb=prod.RHS[p]
                        stk1.push(symb)

                        if(symb.isop==TRUE and symb.lexeme==”]”)
                        {
                                // A -> B [ D | E ]+ F

                                stk1.pop()    //pop ‘]’

                                while((symb= stk1.pop())!=’[‘)
                                        stk2.push(symb)

                                // now stk2 have D | E

                                symb=prod.RHS[p+1]

                                if(symb!=NULL && (symb.lexeme[0]==’*’||
                                                symb.lexeme[0]==’+’ ||
                                                symb.lexeme[0]==’#’))
                                {
                                        p++
                                        if(symb.lexeme[0]==’*’ || symb.lexeme[0]==’+’)
                                        {
                                                // A -> B [ D | E ]* F
                                                // conversions:
                                                // A -> B X2 F
                                                // A -> B F
                                                // X2 -> [D | E] X2
                                                // X2 -> [D | E]
                                                ////////////////////////////
```

```
                                                // A -> B [ D | E ]+ F
                                                // conversions:
                                                // A -> B X2 F
                                                // X2 -> [D | E] X2
                                                // X2 -> [D | E]
```

```
                                                //creating X2
                                                nsymb=CreateNewSymbol(false,”^=!”)       // see param

                                                //creating: X2 -> [D | E]
                                                tprod=new Production()
                                                tprod.LHS=nsymb
                                                tprod.RHS=stk2.GetList()
                                                vlist.add(tprod)

                                                //creating: X2 -> [D | E] X2
                                                tprod=new Production()
                                                tprod.LHS=nsymb
                                                tprod.RHS=stk2.GetList()
                                                tprod.RHS.Add(nsymb)
                                                vlist.add(tprod)

                                                //creating: S -> B F
                                                //for * operator only
                                                if(symb.lexeme[0]==’*’)
                                                {
                                                        prod3=new Production()
                                                        prod3.LHS=prod.LHS
```

```
                        prod3.RHS=stk1.GetList()
                        prod3.RHS.AddList(prod.GetRHSList(p))

                        /*
                         prod.GetRHSList(p) function returns list  of
                        Symbols after the p- index
                        incase of A -> B [ D | E ]* F E and p=6
                        it returns {F E}

                        Now Prod3 has prod3 = A -> B F

                        */

                        vlist.add(prod3)
                }

                stk1.push(nsymb)    // Push X2
                break       // WHY THIS
        }
        else if(symb.lexeme[0]=='#')
        {
                // A -> B [ D % E ]#1#3 F

                // conversions:
                // A -> B X2 F
                // A -> B X2 X2 F
                // A -> B X2 X2 X2 F
                // X2-> [D % E]

                p++ //at number1
                min=prod.RHS[p]
                p++ //at number2
                max= prod.RHS[p]

                //creating X2
                nsymb=CreateNewSymbol(false,"^=!")
                prod2=new Production
                prod2.LHS=nsymb
                prod2.RHS="["+stk2.GetList()+"]"
                vlist.add(prod2)

                left_symb_list  = stk1.GetList()
                right_symb_list = prod.GetRHSList(p+1)

                symb_list=GetAllCasesList(nsymb,min,max)
                /*
                Sym_list contains..

                        1.     X2
                        2.     X2 X2
                        3.     X2 X2 X2
                */
                for(n=0;n<symb_list.size();n++)
                {
                        tprod=new Production()
                        tprod.LHS=prod.LHS
                        tprod.RHS=left_symb_list
                        tprod.RHS.AddList(symb_list[n])
                        tprod.RHS.AddList(right_symb_list)
                        vlist.add(tprod)
                }

                stk1.push(nsymb)
```

```
                                        break

                                } //end else if
                        }
                        else
                        // productions enclosed in [XYZ DS DSD] with no
                        // outer-operator *, + or #
                        {
                                ///////////////////////////X->[[A%[A|B]]|D]
                                //      X -> [A%B]
                                //      X -> A B
                                //      X -> B A

                                nsymb=CreateNewSymbol(false,"^=!")
                                tprod=new Production()
                                tprod.LHS=nsymb
                                tprod.RHS=stk2.GetList()
                                vlist.add(tprod)

                                tprod=new Production()
                                tprod.LHS=nsymb
                                tprod.RHS=stk2.GetListReverse()
                                vlist.add(tprod)

                                prod_list=GetResolvedProdList(tprod)
                                vlist.addList(prod_list)

                                stk1.push(nsymb)
                        }

                }
        prod.RHS=stk1.GetList()
        }//end for

}
```

```
P_Grammar::GramarReader()
{

        **Read First Production Rule**

        While(!eof())
        {
                RawProduction * Aprod;

                wstring LHS = assign LHS String for current production

                Aprod->Lhs = SymbolMap:: getSymbol(LHS)

                For each RHSs
                {
                //   Fdescription are attached with each symbol of the RHSs symbol
                     Aprod->insertSymbol(SymbolMap:: getSymbol(CurrSymbol)
                }
                  Assign probability to the production

                **Read Next Rule**
        }
}
vector<P_CProductions *>  PurifyProductions(vector<RawProductions *> prod)
```

```
{
        P_Cproduction * temp

        for(for each RawProduction [say rprod])
        {
                temp=new P_CProduction();

                temp->LHS = rprod->LHS ;

                temp->SelectionSet = FormatSelectionSet( rprod->SelectionSet )
/*
                Selection sets are formated in Bit format
                Every terminal symbol is taken as 2^n where n is a the terminal symbol number
*/

                for(for each RHS of rprod  [currSymb]  )
                {
                        psymb=new CProductionSymbol();

                        Assign SymbolId

                        Assign IsTerminal that the symbol is terminal or nonterminal

        Assign FdescriptionsIndices for each Fdescription mapped from FdecriptionMap::GetMapFDesc(FDesc)

                        temp->RHS.push_back(psymb);
                }
                prod_table->pushback(temp)
        }

Return prod_table;
}
```

```
void ReadLexicon()
{
//      Read First Entry in wstring

        While(!eof())
        {
          LexicalEntry* ALexEntry = new LexicalEntry;

        //      assign   ALexEntry probably if there.
        //      separate all FDescriptions
                vector<string> FDescriptions = All the FDescriptions of that entry

        for all same entries with only different in Fdescriptions, All Fdescriptions are Ored for these entries and then
        their Fdescriptions are converted to FdescriptionsIndices

        In case of same entries with different Subcat Frames, a vector of indeces of subcat frame is attached with
        Lexical Entry

        Assign Subcat Frames by taking index from
        Byte fdindex = P_F_DescriptionMap::GetIndex(vector<wstring> SubCat)

        ALexEntry.SemForm.Gfindex = fdindex
        /*
        Format of ORed FDescriptions:

        [ [Fdescription1 || Fdescription2] || Fdescription3]

        */

        AlexEntry->FDescriptionsIndices =P_F_DescriptionMap:: GetFDescriptions(FDescriptions)
        Wstring pred = assign pred of the entry
          ALexEntry.SemForm.pred = CsymbolTableManager::getNum(pred)      // How to assign SemForm

        wstring SurfaceForm = assign Surface form

        Hash_Multimap[SurfaceForm]= ALexEntry         // How to insert LexEntry  in HashMap

        //  ReadNextEntry

        }
}
ReadSubcatFrame()
{
/*      File Format      "SubCats.txt"
Subj
Subj    Obj
Subj    Obj     Obj2
*/

Read in SubCatFrames from File "SubCats.txt"

}

GetSubcatFrameIndex(vector<byte> Subcats): byte index
{
}

GetSubcatFrame (byte index): vector<byte> Subcats
{
}
```