Lekta framework practical tutorial

Jose F Quesada & Jose Luis Pro

What is Lekta?

Lekta is a software **framework** oriented to the design and implementation of Natural Language Processing (NLP) related applications. This includes:

- Some specialized and **optimized** modules widely used in NLP applications (tokenizer, parser and so on). You'll never reinvent the wheel anymore.
- A simple and efficient way to define lexicons and grammar rules for any language.
- Early multilingual support for all your applications.
- A set of built-in functions that you'll find useful when implementing your NLP oriented app.
- A programming language to interact with all items above and to define your own functions or procedures.

What is Lekta?

Lekta is a software **framework** oriented to the design and implementation of Natural Language Processing (NLP) related applications. This includes:

- Some specialized and **optimized** modules widely used in NLP applications (tokenizer, parser and so on). You'll never reinvent the wheel anymore.
- A simple and efficient way to define lexicons and grammar rules for any language.
- Early multilingual support for all your applications.
- A set of built-in functions that you'll find useful when implementing your NLP oriented app.
- A programming language to interact with all items above and to define your own functions or procedures.

What is Lekta?

Lekta is a software **framework** oriented to the design and implementation of Natural Language Processing (NLP) related applications. This includes:

- Some specialized and **optimized** modules widely used in NLP applications (tokenizer, parser and so on). You'll never reinvent the wheel anymore.
- A simple and efficient way to define lexicons and grammar rules for any language.
- Early multilingual support for all your applications.
- A set of built-in functions that you'll find useful when implementing your NLP oriented app.
- A programming language to interact with all items above and to define your own functions or procedures.

What is Lekta?

Lekta is a software **framework** oriented to the design and implementation of Natural Language Processing (NLP) related applications. This includes:

- Some specialized and **optimized** modules widely used in NLP applications (tokenizer, parser and so on). You'll never reinvent the wheel anymore.
- A simple and efficient way to define lexicons and grammar rules for any language.
- Early multilingual support for all your applications.
- A set of built-in functions that you'll find useful when implementing your NLP oriented app.
- A programming language to interact with all items above and to define your own functions or procedures.

What is Lekta?

Lekta is a software **framework** oriented to the design and implementation of Natural Language Processing (NLP) related applications. This includes:

- Some specialized and **optimized** modules widely used in NLP applications (tokenizer, parser and so on). You'll never reinvent the wheel anymore.
- A simple and efficient way to define lexicons and grammar rules for any language.
- Early multilingual support for all your applications.
- A set of built-in functions that you'll find useful when implementing your NLP oriented app.
- A programming language to interact with all items above and to define your own functions or procedures.

What is Lekta?

Lekta is a software **framework** oriented to the design and implementation of Natural Language Processing (NLP) related applications. This includes:

- Some specialized and **optimized** modules widely used in NLP applications (tokenizer, parser and so on). You'll never reinvent the wheel anymore.
- A simple and efficient way to define lexicons and grammar rules for any language.
- Early multilingual support for all your applications.
- A set of built-in functions that you'll find useful when implementing your NLP oriented app.
- A programming language to interact with all items above and to define your own functions or procedures.

Basic project setup

A lekta project is composed of a single text file.

This file starts with the keyword “lektaProject” and has, at least, five sections:

```
lektaProject

   projectHead
      <...>

   projectSetup
      <...>

   classModel
      <...>

   lexicalModel forLanguage <...>
      <...>

   grammaticalModel forLanguage <...>
      <...>
```

Basic project setup

A lekta project is composed of a single text file.

This file starts with the keyword "`lektaProject`" and has, at least, five sections:

```
lektaProject

    projectHead
        <...>

    projectSetup
        <...>

    classModel
        <...>

    lexicalModel forLanguage <...>
        <...>

    grammaticalModel forLanguage <...>
        <...>
```

Basic project setup

A lekta project is composed of a single text file.

This file starts with the keyword "`lektaProject`" and has, at least, five sections:

```
lektaProject

    projectHead
        <...>

    projectSetup
        <...>

    classModel
        <...>

    lexicalModel forLanguage <...>
        <...>

    grammaticalModel forLanguage <...>
        <...>
```

AnBm.lkt (1/4)

Example

- We would like to create a very simple lekta project.
- It must be able to recognize sentences in a formal language defined as $A_n B_m$ with $n, m \geq 1$.
- In other words, this language is composed by all the strings that have at least one "a" followed by, at least, one "b".
- a a a a b b b: Correct.
- b b b a a a a: Incorrect.

```
projectHead
   // Languages defined for this project.
   projectLanguageScope : [ anbm ]

   // Output file after compiling.
   projectCompileOutput : ".AnBm.olk"
```

AnBm.lkt (1/4)

Example

- We would like to create a very simple lekta project.
- It must be able to recognize sentences in a formal language defined as $A_n B_m$ with $n, m \geq 1$.
- In other words, this language is composed by all the strings that have at least one "a" followed by, at least, one "b".
- a a a a b b b: Correct.
- b b b a a a a: Incorrect.

```
projectHead
   // Languages defined for this project.
   projectLanguageScope : [ anbm ]

   // Output file after compiling.
   projectCompileOutput : ".AnBm.olk"
```

AnBm.lkt (1/4)

Example

- We would like to create a very simple lekta project.
- It must be able to recognize sentences in a formal language defined as $A_n B_m$ with $n, m \geq 1$.
- In other words, this language is composed by all the strings that have at least one "a" followed by, at least, one "b".
- a a a a b b b: Correct.
- b b b a a a a: Incorrect.

```
projectHead
   // Languages defined for this project.
   projectLanguageScope : [ anbm ]

   // Output file after compiling.
   projectCompileOutput : ".AnBm.olk"
```

AnBm.lkt (1/4)

Example

- We would like to create a very simple lekta project.
- It must be able to recognize sentences in a formal language defined as $A_n B_m$ with $n, m \geq 1$.
- In other words, this language is composed by all the strings that have at least one "a" followed by, at least, one "b".
- `a a a a b b b`: Correct.
- `b b b a a a a`: Incorrect.

```
projectHead
   // Languages defined for this project.
   projectLanguageScope : [ anbm ]

   // Output file after compiling.
   projectCompileOutput : ".AnBm.olk"
```

AnBm.lkt (1/4)

Example

- We would like to create a very simple lekta project.
- It must be able to recognize sentences in a formal language defined as $A_n B_m$ with $n, m \geq 1$.
- In other words, this language is composed by all the strings that have at least one "a" followed by, at least, one "b".
- `a a a a b b b`: Correct.
- `b b b a a a a`: Incorrect.

```
projectHead
   // Languages defined for this project.
   projectLanguageScope : [ anbm ]

   // Output file after compiling.
   projectCompileOutput : ".AnBm.olk"
```

AnBm.lkt (1/4)

Example

- We would like to create a very simple lekta project.
- It must be able to recognize sentences in a formal language defined as $A_n B_m$ with $n, m \geq 1$.
- In other words, this language is composed by all the strings that have at least one “a” followed by, at least, one “b”.
- a a a a b b b: Correct.
- b b b a a a a: Incorrect.

```
projectHead
   // Languages defined for this project.
   projectLanguageScope : [ anbm ]

   // Output file after compiling.
   projectCompileOutput : ".AnBm.olk"
```

AnBm.lkt (2/4)

```
projectSetup
    // Possible output parser types.
    setupParserRoots = S

classModel
    // Definition of all the types needed.
    // Type Void acts as a label.
    classDef:Void ( S, A, B, a, b )

// Lexical model is language dependant
lexicalModel forLanguage anbm
    // Lexicon elements that we must detect.
    // Here a and b act as grammar terminal
        symbols.
    ("a", a)
    ("b", b)
```

AnBm.lkt (3/4)

```
// Grammatical model is language dependant
grammaticalModel forLanguage anbm
    // List of grammar rules
    /* Always context free grammar. Among
       other things, this means that left
       part of the rule must be composed by
       one and only one non-terminal symbol
       */
    (R1: [ S -> a A b B ])
    (R2: [ A -> ])
    (R3: [ A -> a A ])
    (R4: [ B -> ])
    (R5: [ B -> b B ])
```

AnBm.lkt (4/4)

```
// ******************************************************************
//
// Exercise 01: Generator/Recognizer for language AnBm. Where n,m >= 1
//
// ******************************************************************

lektaProject
   projectHead
      projectLanguageScope : [ anbm ]
      projectCompileOutput : ".AnBm.olk"

   projectSetup
      setupParserRoots = S

   classModel
      classDef:Void ( S, A, B, a, b )

   lexicalModel forLanguage anbm
      ("a", a)
      ("b", b)

   grammaticalModel forLanguage anbm
      (R1: [ S -> a A b B ])
      (R2: [ A -> ])
      (R3: [ A -> a A ])
      (R4: [ B -> ])
      (R5: [ B -> b B ])
```

Compiling and executing

- After creating `AnBm.lkt` file we must compile it:
- `$> lektac AnBm.lkt`
- You must see: "`Compilation Succesfully Finished`" message.
- And after that you must create a file for the lekta interpreter, `AnBm.slk`, to test and execute the project:
- `$> synclekta AnBm.slk`

```
// Start lekta engine
LaunchLektaKernel()

// Use recently compiled project
UseProject (ProjectCompile : ".AnBm.olk")

// Options for visualization
DisplayProcessUnderstandingOn

// Start a dialogue with lekta
CreateDialogue()
```

Compiling and executing

- After creating `AnBm.lkt` file we must compile it:
- `$> lektac AnBm.lkt`
- You must see: "`Compilation Succesfully Finished`" message.
- And after that you must create a file for the lekta interpreter, `AnBm.slk`, to test and execute the project:
- `$> synclekta AnBm.slk`

```
// Start lekta engine
LaunchLektaKernel()

// Use recently compiled project
UseProject (ProjectCompile : ".AnBm.olk")

// Options for visualization
DisplayProcessUnderstandingOn

// Start a dialogue with lekta
CreateDialogue()
```

Compiling and executing

- After creating `AnBm.lkt` file we must compile it:
- `$> lektac AnBm.lkt`
- You must see: "`Compilation Succesfully Finished`" message.
- And after that you must create a file for the lekta interpreter, `AnBm.slk`, to test and execute the project:
- `$> synclekta AnBm.slk`

```
// Start lekta engine
LaunchLektaKernel()

// Use recently compiled project
UseProject (ProjectCompile : ".AnBm.olk")

// Options for visualization
DisplayProcessUnderstandingOn

// Start a dialogue with lekta
CreateDialogue()
```

Compiling and executing

- After creating `AnBm.lkt` file we must compile it:
- `$> lektac AnBm.lkt`
- You must see: "`Compilation Succesfully Finished`" message.
- And after that you must create a file for the lekta interpreter, `AnBm.slk`, to test and execute the project:
- `$> synclekta AnBm.slk`

```
// Start lekta engine
LaunchLektaKernel()

// Use recently compiled project
UseProject (ProjectCompile : ".AnBm.olk")

// Options for visualization
DisplayProcessUnderstandingOn

// Start a dialogue with lekta
CreateDialogue()
```

Compiling and executing

- After creating `AnBm.lkt` file we must compile it:
- `$> lektac AnBm.lkt`
- You must see: "`Compilation Succesfully Finished`" message.
- And after that you must create a file for the lekta interpreter, `AnBm.slk`, to test and execute the project:
- `$> synclekta AnBm.slk`

```
// Start lekta engine
LaunchLektaKernel()

// Use recently compiled project
UseProject (ProjectCompile : ".AnBm.olk")

// Options for visualization
DisplayProcessUnderstandingOn

// Start a dialogue with lekta
CreateDialogue()
```

Session 01: Exercise 01
AnBm

Grammar rules syntax

```
(<rule_label>:
    [ <symbol> -> <list_of_symbols> ]
    {
        ...
        <commands>
        ...
    }
)
```

- <rule_label> Only useful for readability and debugging.
- <symbol> A non-terminal symbol.
- <list_of_symbols> List of terminal and non-terminal symbols needed for the triggering of this rule.
- <commands> Commands to be executed when this rule is triggered.

Grammar rules syntax

```
(<rule_label>:
    [ <symbol> -> <list_of_symbols> ]
    {
        ...
        <commands>
        ...
    }
)
```

- `<rule_label>` Only useful for readability and debugging.
- `<symbol>` A non-terminal symbol.
- `<list_of_symbols>` List of terminal and non-terminal symbols needed for the triggering of this rule.
- `<commands>` Commands to be executed when this rule is triggered.

Special features: Optional symbol (?)

Rule R2 can be interpreted as a mandatory 'a' followed (or not) by 'A'.

```
(R1: [ S -> A? B? ])
(R2: [ A -> a A? ])
```

```
// These rules are expanded into standard rules
(R1: [ S -> A B ])
(R1: [ S -> B ])
(R1: [ S -> A ])
(R1: [ S -> ])

(R2: [ A -> a A ])
(R2: [ A -> a ])
```

Special features: Optional symbol (?)

Rule `R2` can be interpreted as a mandatory 'a' followed (or not) by 'A'.

```
(R1: [ S -> A? B? ])
(R2: [ A -> a A? ])
```

```
// These rules are expanded into standard rules
(R1: [ S -> A B ])
(R1: [ S -> B ])
(R1: [ S -> A ])
(R1: [ S -> ])

(R2: [ A -> a A ])
(R2: [ A -> a ])
```

Special features: Or symbol (|)

Rule `R1` can be interpreted as a mandatory ‘A’ followed by ‘B’, ‘C’ or ‘D’.

```
(R1: [ S -> A < B | C | D > ])
```

```
// These rules are expanded into standard rules
(R1: [ S -> A B ])
(R1: [ S -> A C ])
(R1: [ S -> A D ])
```

Special symbols can be combined:

```
(R1: [ S -> A < B | C >? ])

(R1: [ S -> A ])
(R1: [ S -> A B ])
(R1: [ S -> A C ])
```

Special features: Or symbol (|)

Rule R1 can be interpreted as a mandatory 'A' followed by 'B', 'C' or 'D'.

```
(R1: [ S -> A < B | C | D > ])
```

```
// These rules are expanded into standard rules
(R1: [ S -> A B ])
(R1: [ S -> A C ])
(R1: [ S -> A D ])
```

Special symbols can be combined:

```
(R1: [ S -> A < B | C >? ])

(R1: [ S -> A ])
(R1: [ S -> A B ])
(R1: [ S -> A C ])
```

Special features: Or symbol (|)

Rule `R1` can be interpreted as a mandatory ‘A’ followed by ‘B’, ‘C’ or ‘D’.

```
(R1: [ S -> A < B | C | D > ])
```

```
// These rules are expanded into standard rules
(R1: [ S -> A B ])
(R1: [ S -> A C ])
(R1: [ S -> A D ])
```

Special symbols can be combined:

```
(R1: [ S -> A < B | C >? ])

(R1: [ S -> A ])
(R1: [ S -> A B ])
(R1: [ S -> A C ])
```

Special features: Free order symbol (%)

Rule R1 can be interpreted as a mandatory 'A' followed by 'B', 'C' and 'D' in any order.

```
(R1: [ S -> A < B % C % D > ])
```

```
// These rules are expanded into standard rules
(R1: [ S -> A B C D ])
(R1: [ S -> A B D C ])
(R1: [ S -> A C B D ])
(R1: [ S -> A C D B ])
(R1: [ S -> A D B C ])
(R1: [ S -> A D C B ])
```

When combining several special symbols we must take into account exponentially growing in the number of rules.

```
// 12 Rules when expanding
(R1: [ S -> A < B % C % < D | E > > ])
```

Special features: Free order symbol (%)

Rule R1 can be interpreted as a mandatory 'A' followed by 'B', 'C' and 'D' in any order.

```
(R1: [ S -> A < B % C % D > ])
```

```
// These rules are expanded into standard rules
(R1: [ S -> A B C D ])
(R1: [ S -> A B D C ])
(R1: [ S -> A C B D ])
(R1: [ S -> A C D B ])
(R1: [ S -> A D B C ])
(R1: [ S -> A D C B ])
```

When combining several special symbols we must take into account exponentially growing in the number of rules.

```
// 12 Rules when expanding
(R1: [ S -> A < B % C % < D | E > > ])
```

Special features: Free order symbol (%)

Rule `R1` can be interpreted as a mandatory ‘A’ followed by ‘B’, ‘C’ and ‘D’ in any order.

```
(R1: [ S -> A < B % C % D > ])
```

```
// These rules are expanded into standard rules
(R1: [ S -> A B C D ])
(R1: [ S -> A B D C ])
(R1: [ S -> A C B D ])
(R1: [ S -> A C D B ])
(R1: [ S -> A D B C ])
(R1: [ S -> A D C B ])
```

When combining several special symbols we must take into account exponentially growing in the number of rules.

```
// 12 Rules when expanding
(R1: [ S -> A < B % C % < D | E > > ])
```

Special features: Left and right limits (&[+^] and [+$]&)

Rule R1 will only be triggered if there is nothing to the left of 'A' expression:

```
(R1: [ S -> &[+^] A B ])
```

Rule R1 will only be triggered if there is nothing to the right of 'B' expression:

```
(R1: [ S -> A B [+$]& ])
```

Rule R1 will only be triggered if there is nothing to the left of 'A' expression and nothing to the right of 'B' expression:

```
(R1: [ S -> &[+^] A B [+$]& ])
```

Special features: Left and right limits (&[+^] and [+$]&)

Rule R1 will only be triggered if there is nothing to the left of 'A' expression:

```
(R1: [ S -> &[+^] A B ])
```

Rule R1 will only be triggered if there is nothing to the right of 'B' expression:

```
(R1: [ S -> A B [+$]& ])
```

Rule R1 will only be triggered if there is nothing to the left of 'A' expression and nothing to the right of 'B' expression:

```
(R1: [ S -> &[+^] A B [+$]& ])
```

Special features: Left and right limits (&[+^] and [+$]&)

Rule R1 will only be triggered if there is nothing to the left of 'A' expression:

```
(R1: [ S -> &[+^] A B ])
```

Rule R1 will only be triggered if there is nothing to the right of 'B' expression:

```
(R1: [ S -> A B [+$]& ])
```

Rule R1 will only be triggered if there is nothing to the left of 'A' expression and nothing to the right of 'B' expression:

```
(R1: [ S -> &[+^] A B [+$]& ])
```

Session 01: Exercise 02
AnBm with optional parameters

Session 01: Exercise 03
AnBmCn

Type specification in Lekta

- At the very beginning we have not any type in our project.
- So you must create all the types you may need (with `classDef` keyword).
- To create new types we have metatypes (or type creators) in Lekta:
  1. `ElementBool`: Boolean metatype.
  2. `ElementInt`: Integer numbers metatype.
  3. `ElementReal`: Real numbers metatype.
  4. `ElementLiteral`: Strings of characters.
  5. `ElementMessage`: Messages.
  6. `ElementRange`: Enumerate type.
  7. `StructureBatch`: Sequence of elements of the same type.
  8. `StructureComplex`: Structure of elements of any type.

Type specification in Lekta

- At the very beginning we have not any type in our project.
- So you must create all the types you may need (with `classDef` keyword).
- To create new types we have metatypes (or type creators) in Lekta:
  1. `ElementBool`: Boolean metatype.
  2. `ElementInt`: Integer numbers metatype.
  3. `ElementReal`: Real numbers metatype.
  4. `ElementLiteral`: Strings of characters.
  5. `ElementMessage`: Messages.
  6. `ElementRange`: Enumerate type.
  7. `StructureBatch`: Sequence of elements of the same type.
  8. `StructureComplex`: Structure of elements of any type.

Type specification in Lekta

- At the very beginning we have not any type in our project.
- So you must create all the types you may need (with `classDef` keyword).
- To create new types we have metatypes (or type creators) in Lekta:
  1. `ElementBool`: Boolean metatype.
  2. `ElementInt`: Integer numbers metatype.
  3. `ElementReal`: Real numbers metatype.
  4. `ElementLiteral`: Strings of characters.
  5. `ElementMessage`: Messages.
  6. `ElementRange`: Enumerate type.
  7. `StructureBatch`: Sequence of elements of the same type.
  8. `StructureComplex`: Structure of elements of any type.

Type specification in Lekta

- So you **can't** declare a variable of type `ElementInt`.
- But you **can** create a type (let's say "`integer`") with the metatype `ElementInt` and declare a variable of type `integer`.

Incorrect Example

```
...
ElementInt i <- 5;
...
```

Correct Example

```
classDef:ElementInt ( integer )
...
integer i <- 5;
...
```

Type specification in Lekta

- So you **can't** declare a variable of type `ElementInt`.
- But you **can** create a type (let's say "`integer`") with the metatype `ElementInt` and declare a variable of type `integer`.

Incorrect Example

```
...
ElementInt i <- 5;
...
```

Correct Example

```
classDef:ElementInt ( integer )
...
integer i <- 5;
...
```

Type specification in Lekta

- So you **can't** declare a variable of type `ElementInt`.
- But you **can** create a type (let's say "`integer`") with the metatype `ElementInt` and declare a variable of type `integer`.

Incorrect Example

```
...
ElementInt i <- 5;
...
```

Correct Example

```
classDef:ElementInt ( integer )
...
integer i <- 5;
...
```

Type specification in Lekta

- So you **can't** declare a variable of type `ElementInt`.
- But you **can** create a type (let's say "`integer`") with the metatype `ElementInt` and declare a variable of type `integer`.

Incorrect Example

```
...
ElementInt i <- 5;
...
```

Correct Example

```
classDef:ElementInt ( integer )
...
integer i <- 5;
...
```

Type specification in Lekta

- Sometimes is useful to create some basic types in the beginning.
- For example, if you are Java fan:

```
classDef:ElementInt     ( int )
classDef:ElementBool    ( boolean )
classDef:ElementLiteral ( String )
...
int i <- 5;
boolean flag <- False;
String s <- 'this is a string';
...
```

Type specification in Lekta

- Sometimes is useful to create some basic types in the beginning.
- For example, if you are Java fan:

```
classDef:ElementInt      ( int )
classDef:ElementBool     ( boolean )
classDef:ElementLiteral  ( String )
...
int i <- 5;
boolean flag <- False;
String s <- 'this is a string';
...
```

Complex structures

```
classDef:ElementInt      (Counter)
classDef:ElementBool     (Flag)
classDef:ElementLiteral (Expression)
classDef:StructureComplex
(
    ExampleStructure:
    (
        Counter,
        Flag,
        Expression
    )
)
...
ExampleStructure a;
a.Counter <- 5;
a.Flag <- True;
a.Expression <- 'this is a string';
...
```

- Previous exercises only make some syntactic analysis of the language.
- So we can only recognize valid sentences in that language (paser output is a void 'S').
- But we want now to do different things with that sentences.
- So, how can we provide some semantic content to AnBm language?
- The only reasonable semantic content is to have 'n' and 'm' values associated with 'S' structure.

```
classDef:ElementInt ( counterN, counterM )
classDef:StructureComplex ( S:( counterN, counterM ) )
classDef:StructureComplex ( A:( counterN ) )
classDef:StructureComplex ( B:( counterM ) )
classDef:Void ( a, b )
```

- Previous exercises only make some syntactic analysis of the language.
- So we can only recognize valid sentences in that language (paser output is a void 'S').
- But we want now to do different things with that sentences.
- So, how can we provide some semantic content to AnBm language?
- The only reasonable semantic content is to have 'n' and 'm' values associated with 'S' structure.

```
classDef:ElementInt ( counterN, counterM )
classDef:StructureComplex ( S:( counterN, counterM ) )
classDef:StructureComplex ( A:( counterN ) )
classDef:StructureComplex ( B:( counterM ) )
classDef:Void ( a, b )
```

- Previous exercises only make some syntactic analysis of the language.
- So we can only recognize valid sentences in that language (paser output is a void 'S').
- But we want now to do different things with that sentences.
- So, how can we provide some semantic content to AnBm language?
- The only reasonable semantic content is to have 'n' and 'm' values associated with 'S' structure.

```
classDef:ElementInt ( counterN, counterM )
classDef:StructureComplex ( S:( counterN, counterM ) )
classDef:StructureComplex ( A:( counterN ) )
classDef:StructureComplex ( B:( counterM ) )
classDef:Void ( a, b )
```

- Previous exercises only make some syntactic analysis of the language.
- So we can only recognize valid sentences in that language (paser output is a void 'S').
- But we want now to do different things with that sentences.
- So, how can we provide some semantic content to AnBm language?
- The only reasonable semantic content is to have 'n' and 'm' values associated with 'S' structure.

```
classDef:ElementInt ( counterN, counterM )
classDef:StructureComplex ( S:( counterN, counterM ) )
classDef:StructureComplex ( A:( counterN ) )
classDef:StructureComplex ( B:( counterM ) )
classDef:Void ( a, b )
```

- Previous exercises only make some syntactic analysis of the language.
- So we can only recognize valid sentences in that language (paser output is a void 'S').
- But we want now to do different things with that sentences.
- So, how can we provide some semantic content to AnBm language?
- The only reasonable semantic content is to have 'n' and 'm' values associated with 'S' structure.

```
classDef:ElementInt ( counterN, counterM )
classDef:StructureComplex ( S:( counterN, counterM ) )
classDef:StructureComplex ( A:( counterN ) )
classDef:StructureComplex ( B:( counterM ) )
classDef:Void ( a, b )
```

```
(R1: [ S -> A B ] {
       ^.counterN <- #1.counterN;
       ^.counterM <- #2.counterM; } )

(R2: [ A -> a A ] {
       ^.counterN <- 1 + #2.counterN; } )

(R3: [ A -> a ] {
       ^.counterN <- 1; } )

(R4: [ B -> b B ] {
       ^.counterM <- 1 + #2.counterM; } )

(R5: [ B -> b ] {
       ^.counterM <- 1; } )
```

Note special syntax:

- ^ stands for left side term generated by the rule (upper node in parsing tree).
- #1 stands for the first term in the right side of the rule.
- #2 stands for the second term in the right side of the rule.
- #N stands for the nth term in the right side of the rule.

Session 01: Exercise 04
AnBn

Session 02: Exercise 01
English lexicon and grammar

Files inclusion

```
lektaProject

   projectHead
      projectLanguageScope : [ en ]
      projectCompileOutput : ".Numbers.olk"

   projectSetup
      setupParserRoots = Number

   classModel
      #Include "NumberTypes.lkt"

   lexicalModel forLanguage en
      #Include "NumberEnglishLexicon.lkt"

   grammaticalModel forLanguage en
      #Include "NumberEnglishGrammar.lkt"
```

Function and procedure declaration

```
classDef:ElementInt ( integer )
classDef:ElementBool ( bool )
// Templates
<ouput_type> function_name(<parameter_list>) {
    ...
}
procedure function_name(<parameter_list>) {
    ...
}

// Examples
bool f1(integer i) {
    ...
}
procedure f2(integer i) {
    ...
}
procedure f3() { // Not "void" keyword
    ...
}
```

Comments

```
// This is a mono-line comment
/* This is a multi-line comment
       with some commented lines   */
```

Arithmetic operators

```
a <- b + c; // Addition
a <- b - c; // Subtraction
a <- b * c; // Multiplication
a <- b / c; // Division
a++;        // Post-autoincrement
++a;        // Pre-autoincrement
a--;        // Post-autodecrement
--a;        // Pre-autodecrement
```

Comments

```
// This is a mono-line comment
/* This is a multi-line comment
      with some commented lines  */
```

Arithmetic operators

```
a <- b + c; // Addition
a <- b - c; // Subtraction
a <- b * c; // Multiplication
a <- b / c; // Division
a++;        // Post-autoincrement
++a;        // Pre-autoincrement
a--;        // Post-autodecrement
--a;        // Pre-autodecrement
```

Comparation operators

```
a >  b  // Greater
a >= b  // Greater or equal
a <  b  // Less
a <= b  // Less or equal
a == b  // Equal
a != b  // Not equal
```

Boolean operators

```
a && b  // And
a || b  // Or
!! a    // Not
```

Comparation operators

```
a >  b  // Greater
a >= b  // Greater or equal
a <  b  // Less
a <= b  // Less or equal
a == b  // Equal
a != b  // Not equal
```

Boolean operators

```
a && b  // And
a || b  // Or
!! a    // Not
```

Lazy evaluation

```
boolean f1()
{
    // Writes a message to standard output
    SpyMessage("Message from f1");
    return False;
}

boolean f2()
{
    SpyMessage("Message from f2");
    return True;
}

procedure testingLazyEvaluation()
{
    boolean b;

    b <- f1() && f2(); // "Message from f1"
    b <- f2() || f1(); // "Message from f2"
}
```

Programming structures: if...else if...else

```
if(month == 1)            { ret <- 'January';   }
else if(month == 2)       { ret <- 'February';  }
else if(month == 3)       { ret <- 'March';     }
else if(month == 4)       { ret <- 'April';     }
else if(month == 5)       { ret <- 'May';       }
else if(month == 6)       { ret <- 'June';      }
else if(month == 7)       { ret <- 'July';      }
else if(month == 8)       { ret <- 'August';    }
else if(month == 9)       { ret <- 'September'; }
else if(month == 10)      { ret <- 'October';   }
else if(month == 11)      { ret <- 'November';  }
else                      { ret <- 'December';  }
```

Programming structures: switch

```
switch (month)
{
    case  1 { ret <- 'January';}
    case  2 { ret <- 'February';}
    case  3 { ret <- 'March';}
    case  4 { ret <- 'April';}
    case  5 { ret <- 'May';}
    case  6 { ret <- 'June';}
    case  7 { ret <- 'July';}
    case  8 { ret <- 'August';}
    case  9 { ret <- 'September';}
    case 10 { ret <- 'October';}
    case 11 { ret <- 'November';}
    default { ret <- 'December';}
}
```

Programming structures: cond

```
cond
{
    (month == 1)  { ret <- 'January';   }
    (month == 2)  { ret <- 'February';  }
    (month == 3)  { ret <- 'March';     }
    (month == 4)  { ret <- 'April';     }
    (month == 5)  { ret <- 'May';       }
    (month == 6)  { ret <- 'June';      }
    (month == 7)  { ret <- 'July';      }
    (month == 8)  { ret <- 'August';    }
    (month == 9)  { ret <- 'September'; }
    (month == 10) { ret <- 'October';   }
    (month == 11) { ret <- 'November';  }
    default       { ret <- 'December';  }
}
```

Programming structures: loops

"While" loop

```
integer position,size;
...
position <- 1;
while (position <= size) {
   <...>
   position++;
}
```

"For" loop

```
integer position,size;
...
for (position <- 1; position <= size; position++) {
   <...>
}
```

Programming structures: loops

"While" loop

```
integer position,size;
...
position <- 1;
while (position <= size) {
    <...>
    position++;
}
```

"For" loop

```
integer position,size;
...
for (position <- 1; position <= size; position++) {
    <...>
}
```

Built-in functions

Mathematics

```
integer Max(integer n1, integer n2);
integer Min(integer n1, integer n2);
integer Ceiling(real r);
integer Floor(real r);
integer Round(real r);
integer Abs(integer n);
integer Modulo(integer dividend, integer divisor);
real Sqrt(real r);
real Pow(real r);
real Exp(real r);
real Log10(real r);
real LogN(real r);
real Sin(real r);
real Cos(real r);
real Tan(real);
integer Random(integer from, integer to);
```

Built-in functions

Date & time

```
integer ClockAskYear();
integer ClockAskMonth();
integer ClockAskDayOfTheMonth();
integer ClockAskDayOfTheWeek();
integer ClockAskHour();
integer ClockAskMinute();
integer ClockAskSecond();
```

Atomic types transformations

```
bool ShapeToBool();
integer ShapeToInt();
real ShapeToReal();
string ShapeToLiteral(); // Same as ShapeToString();
message ShapeToMessage();
range ShapeToRange();
```

Built-in functions

Date & time

```
integer ClockAskYear();
integer ClockAskMonth();
integer ClockAskDayOfTheMonth();
integer ClockAskDayOfTheWeek();
integer ClockAskHour();
integer ClockAskMinute();
integer ClockAskSecond();
```

Atomic types transformations

```
bool ShapeToBool();
integer ShapeToInt();
real ShapeToReal();
string ShapeToLiteral(); // Same as ShapeToString();
message ShapeToMessage();
range ShapeToRange();
```

Built-in functions

Literal functions

```
string LiteralConvertLower(string in);
string LiteralConvertUpper(string in);
string LiteralConcat(string in1, string in2);
string LiteralSubstitution
       (string in, string from, string to);
string LiteralGlobalSubstitution
       (string in, string from, string to);
integer LiteralSize(string in);
string LiteralPositionValue(string in, integer pos);
string LiteralSearch(string in, string toLookFor);
bool LiteralIncluded(string toLookFor, string in);
string SubLiteral
       (string in, integer from, integer to);
```

Built-in functions

"Filled" and "Devoid"

```
classDef:ElementInt( f1, f2 )
classDef:StructureComplex( F: (f1, f2) )
...

F f;
Filled(f);     // False
Devoid(f);     // True

f.f1 <- 5;
Filled(f);     // True
Devoid(f);     // False
Filled(f.f1); // True
Devoid(f.f1); // False
Filled(f.f2); // False
Devoid(f.f2); // True

if(f)     { <...> } // Same as Filled(f)
if(!! f) { <...> } // Same as Devoid(f)
```

Lexicon

```
classDef:Void( one, two )
classDef:ElementInt( NumberValue )
classDef:StructureComplex( Number: (NumberValue) )

...

("one", one)
("two", two)

("one", NumberValue, 1)
("two", NumberValue, 2)

("one", Number, (NumberValue: 1))
("two", Number, (NumberValue: 2))
```

Session 02: Exercise 02 Grammar for parameter extraction - Numbers in english

Assignment operator

```
classDef:ElementInt( f1, f2, f3, f4 )
classDef:StructureComplex( S: (f1, f2, f3, f4) )

...

S a;
a.f2 <- 2;
a.f4 <- 4;

S b;
b.f3 <- 3;
b.f4 <- 4;
```

$$\text{a:} \begin{bmatrix} \texttt{f2:} & 2 \\ \texttt{f4:} & 4 \end{bmatrix} \qquad \text{b:} \begin{bmatrix} \texttt{f3:} & 3 \\ \texttt{f4:} & 4 \end{bmatrix}$$

`a <- b;`

$$\text{a:} \begin{bmatrix} \texttt{f3:} & 3 \\ \texttt{f4:} & 4 \end{bmatrix} \qquad \text{b:} \begin{bmatrix} \texttt{f3:} & 3 \\ \texttt{f4:} & 4 \end{bmatrix}$$

Overwrite operator

```
classDef:ElementInt( f1, f2, f3, f4 )
classDef:StructureComplex( S: (f1, f2, f3, f4) )

...

S a;
a.f2 <- 2;
a.f4 <- 4;

S b;
b.f3 <- 3;
b.f4 <- 4;
```

a: $\begin{bmatrix} \texttt{f2:} & 2 \\ \texttt{f4:} & 4 \end{bmatrix}$ b: $\begin{bmatrix} \texttt{f3:} & 3 \\ \texttt{f4:} & 4 \end{bmatrix}$

a <| b;

a: $\begin{bmatrix} \texttt{f2:} & 2 \\ \texttt{f3:} & 3 \\ \texttt{f4:} & 4 \end{bmatrix}$ b: $\begin{bmatrix} \texttt{f3:} & 3 \\ \texttt{f4:} & 4 \end{bmatrix}$

Overwrite operator

```
classDef:ElementInt( f1, f2, f3, f4 )
classDef:StructureComplex( S: (f1, f2, f3, f4) )

...

S a;
a.f2 <- 2;
a.f4 <- 4;

S b;
b.f3 <- 3;
b.f4 <- 5;
```

a: $\begin{bmatrix} \texttt{f2:} & 2 \\ \texttt{f4:} & 4 \end{bmatrix}$ b: $\begin{bmatrix} \texttt{f3:} & 3 \\ \texttt{f4:} & \cancel{4}5 \end{bmatrix}$

`a <| b;`

a: $\begin{bmatrix} \texttt{f2:} & 2 \\ \texttt{f3:} & 3 \\ \texttt{f4:} & \cancel{4}5 \end{bmatrix}$ b: $\begin{bmatrix} \texttt{f3:} & 3 \\ \texttt{f4:} & \cancel{4}5 \end{bmatrix}$

Unification operator

```
classDef:ElementInt( f1, f2, f3, f4 )
classDef:StructureComplex( S: (f1, f2, f3, f4) )

...

S a;
a.f2 <- 2;
a.f4 <- 4;

S b;
b.f3 <- 3;
b.f4 <- 4;
```

a: $\begin{bmatrix} \texttt{f2:} & 2 \\ \texttt{f4:} & 4 \end{bmatrix}$ b: $\begin{bmatrix} \texttt{f3:} & 3 \\ \texttt{f4:} & 4 \end{bmatrix}$

`a <& b;`

a: $\begin{bmatrix} \texttt{f2:} & 2 \\ \texttt{f3:} & 3 \\ \texttt{f4:} & 4 \end{bmatrix}$ b: $\begin{bmatrix} \texttt{f3:} & 3 \\ \texttt{f4:} & 4 \end{bmatrix}$

Unification operator

```
classDef:ElementInt( f1, f2, f3, f4 )
classDef:StructureComplex( S: (f1, f2, f3, f4) )

...

S a;
a.f2 <- 2;
a.f4 <- 4;

S b;
b.f3 <- 3;
b.f4 <- 5;
```

a: $\begin{bmatrix} \texttt{f2:} & 2 \\ \texttt{f4:} & 4 \end{bmatrix}$ b: $\begin{bmatrix} \texttt{f3:} & 3 \\ \texttt{f4:} & \cancel{4}5 \end{bmatrix}$

`a <& b;`

a: $[\ ]$ `Fail();` b: $\begin{bmatrix} \texttt{f3:} & 3 \\ \texttt{f4:} & \cancel{4}5 \end{bmatrix}$

Session 03: Exercise 01
AnBn with unification operator

Session 03: Exercise 02
AnBnCn

Metatype ElementRange

- Used to create enumerated types.
- Hence, the variables belonging to the created type must be equal to one of the values that have been predefined in it.

Examples

```
classDef:ElementRange (
   CompassDirection: { 'north', 'east', 'south', 'west' }
)

classDef:ElementRange (
   Number: { 'singular', 'plural' }
)

classDef:ElementRange (
   Person:{ '1st', '2nd', '3rd' }
)
```

Metatype ElementRange

- Used to create enumerated types.
- Hence, the variables belonging to the created type must be equal to one of the values that have been predefined in it.

Examples

```
classDef:ElementRange (
   CompassDirection: { 'north', 'east', 'south', 'west' }
)

classDef:ElementRange (
   Number: { 'singular', 'plural' }
)

classDef:ElementRange (
   Person:{ '1st', '2nd', '3rd' }
)
```

Metatype ElementRange

- Used to create enumerated types.
- Hence, the variables belonging to the created type must be equal to one of the values that have been predefined in it.

Examples

```
classDef:ElementRange (
   CompassDirection: { 'north', 'east', 'south', 'west' }
)

classDef:ElementRange (
   Number: { 'singular', 'plural' }
)

classDef:ElementRange (
   Person:{ '1st', '2nd', '3rd' }
)
```

Metatype Synonym

- Used to create types with exactly the same structure as other previously created type.

Example

```
classDef:StructureComplex
(
   Agreement:
   (
      Number,
      Person
   )
)

classDef:Synonym
(
   S, NP, VP, det, noun, verb =
   Agreement
)
```

Metatype Synonym

- Used to create types with exactly the same structure as other previously created type.

Example

```
classDef:StructureComplex
(
    Agreement:
    (
        Number,
        Person
    )
)

classDef:Synonym
(
    S, NP, VP, det, noun, verb =
    Agreement
)
```

Session 03: Exercise 03
Agreement in english natural language

Grammar rules levels

- Sometimes, ambiguities in grammar rules are impossible to avoid.
- For example, when we define operations with associative property.

Example: Mathematical expressions parser (classModel)

```
classDef:ElementRange (
   Operator: {
      '+', '-', '*', '/' } )

classDef:StructureComplex (
   Expression: (
      Operator,
      LeftExpression,
      RightExpression ) )

classDef:Synonym (
   LeftExpression, RightExpression = Expression )

classDef:Void( lexAdd )
```

Grammar rules levels

- Sometimes, ambiguities in grammar rules are impossible to avoid.
- For example, when we define operations with associative property.

Example: Mathematical expressions parser (classModel)

```
classDef:ElementRange (
   Operator: {
      '+', '-', '*', '/' } )

classDef:StructureComplex (
   Expression: (
      Operator,
      LeftExpression,
      RightExpression ) )

classDef:Synonym (
   LeftExpression, RightExpression = Expression )

classDef:Void( lexAdd )
```

Grammar rules levels

Example: Mathematical expressions parser (lexicalModel)

```
// This is the lexicon related with addition operation

// two+two
setupTokenizerPunctuation   ("+", lexAdd)

// two + two
("+",         lexAdd)

// two plus two
("plus",      lexAdd)

// two and two
("and",       lexAdd)

// two added to two
("added to", lexAdd)
```

Grammar rules levels

Example: Mathematical expressions parser (grammaticalModel)

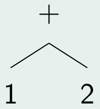
```
(R1: [ Expression -> Number ])

(R2: [ Expression -> Expression lexAdd Expression ]
    {
        ^.Operator <- '+';
        ^.LeftExpression <- #1;
        ^.RightExpression <- #3;
    }
)
```

- Let's assume that we have included lexical and grammatical "Number" (previous exercise).
- So we can build some parser trees.

Grammar rules levels

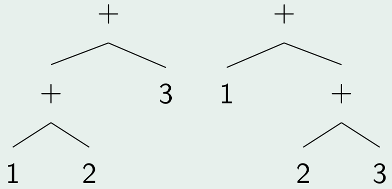
Example: One addition

$$1 + 2$$



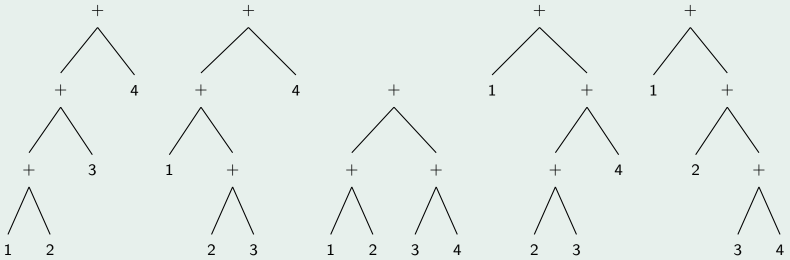
Example: Two additions

$$1 + 2 + 3$$



Example: Three additions

$$1 + 2 + 3 + 4$$



Example: "n" additions

Asimptotically, number of trees grows as n order Catalan number $(C_n)^*$:

$$C_n \sim \frac{4^n}{n^{3/2}} \texttt{ very quick!}$$

(*) for you, math freaks: https://en.wikipedia.org/wiki/Catalan_number

Grammar rules levels

So we must provide some kind of precedence order or priority in order to avoid such ambiguities: Levels in grammar rules.

```
// This rule produces an "Expression term of level 0"
// 0 is the default value so these two rules are equivalent
( R1: [ Expression/0 -> Number ] )
( R1: [ Expression -> Number ] )

// This rule produces an 1-level Expression and needs
// a 0-level (or lower) Expression and 1-level (or lower)
// Expression in order to be triggered:
( R2: [ Expression/1 -> Expression/0 lexAdd Expression/1 ]
    {
        ^.Operator <- '+';
        ^.LeftExpression <- #1;
        ^.RightExpression <- #3;
    }
)
```

Grammar rules levels

```
(R2:[ Expression/1 -> Expression/0 lexAdd Expression/1 ])
```

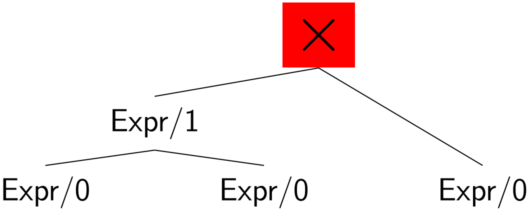
Expr/0 Expr/0 Expr/0

Grammar rules levels

```
(R2:[ Expression/1 -> Expression/0 lexAdd Expression/1 ])
```

Grammar rules levels

```
(R2:[ Expression/1 -> Expression/0 lexAdd Expression/1 ])
```



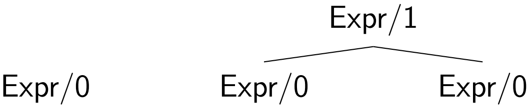
Grammar rules levels

```
(R2:[ Expression/1 -> Expression/0 lexAdd Expression/1 ])
```

Expr/0 Expr/0 Expr/0

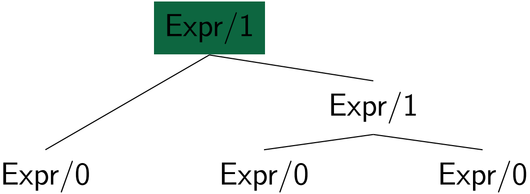
Grammar rules levels

```
(R2:[ Expression/1 -> Expression/0 lexAdd Expression/1 ])
```

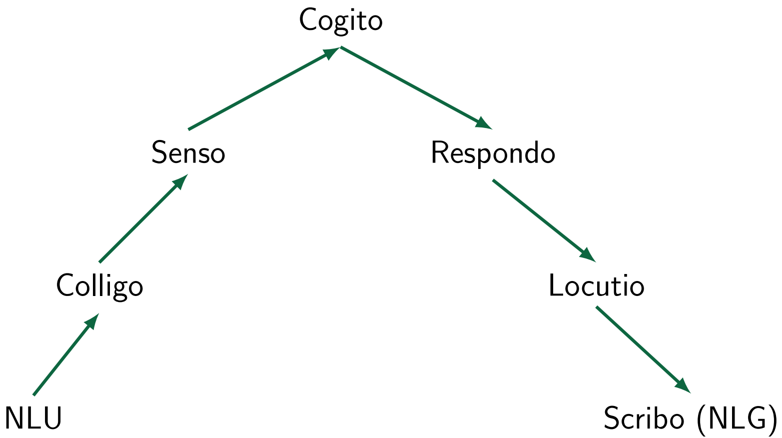


Grammar rules levels

```
(R2:[ Expression/1 -> Expression/0 lexAdd Expression/1 ])
```

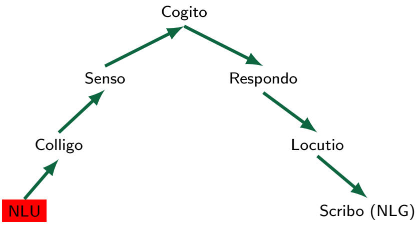


Lekta dialogue manager structure



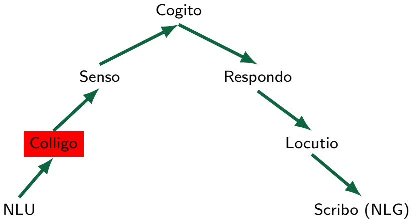
NLU: Natural Language Understanding

- Its main objective is to convert user proference in two or more root types (defined in `setupParserRoots`).
- For example "`Expression`" if we want a mathematical expressions parser.
- Uses defined lexicon and grammar rules.
- All previous examples and exercises are included in this stage.



Colligo: From *latin*, to gather

- It can be used to join two or more parser roots.
- For example: Two consecutives greetings can be merged into only one (`U: Hello, good morning! ...`).
- Possibly in the future this stage will be transformed in some high-level grammar scheme.



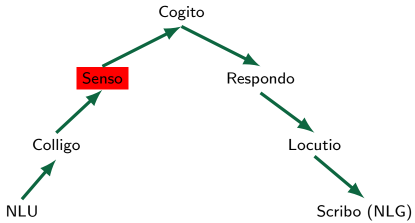
Colligo: From *latin*, to gather

Colligo rules template: ColligoSchemata section

```
(ColligoScheme <rule_name> : [ in_1 ... in_n >> output ]
    ColligoCapture {
        // Preconditions
        <condition_1> &&
        <condition_2> &&
        ...
    }
    ColligoAction {
        // Actions to be executed when triggered
        <command_1>;
        <command_2>;
        ...
        // ^OBJSENSO stands for output term.
        // #OBJCOLLIGO-1 stands for first input term.
        // #OBJCOLLIGO-2 stands for second input term.
        // #OBJCOLLIGO-N stands for nth input term.
    }
)
```

Senso: From *latin*, to sense, to feel

- It corresponds to the sensory part of the use of language.
- Used to filter useless proferences to the dialogue manager.
- Usually this stage writes non-filtered user proferences directly into "mindboard".
- So we can use some rule of this stage to initialize context and mind structures.



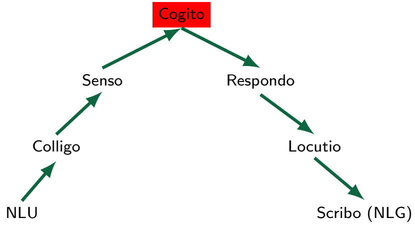
Senso: From *latin*, to sense, to feel

Senso rules template: SensoSchemata section

```
(SensoScheme <rule_name> : [ in_1 ... in_n ]
   SensoCapture {
      // Preconditions
      <condition_1> &&
      <condition_2> &&
      ...
   }
   SensoAction {
      // Actions to be executed when triggered
      <command_1>;
      <command_2>;
      ...
      // #OBJSENSO-1 stands for first input term.
      // #OBJSENSO-2 stands for second input term.
      // #OBJSENSO-N stands for nth input term.
   }
)
```

Cogito: From *latin*, to think

- Its the module that simulates human thinking, context-dependant pragmatics, and logic reasoning.
- It uses some global data structures (the only ones in Lekta) that any other module can read and even write.



Cogito: From *latin*, to think

Mindboard structures definition: `conversationalModel` section

```
MindBoardStructure: {
    ( <name_of_field_1> / <field_type_1> )
    ...
    ( <name_of_field_n> / <field_type_n> )
}
```

Mindboard structures example

```
MindBoardStructure: {
    ( Counter / integer )
    ( Error / bool )
}
...
// To access these fields:
$MINDBOARD@Counter <- 0;
$MINDBOARD@Error <- False;
```

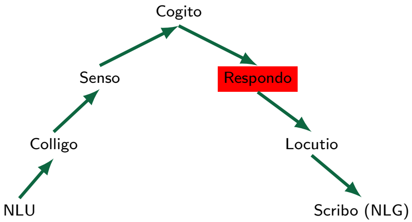
Cogito: From *latin*, to think

Cogito rules template: CogitoSchemata section

```
(CogitoScheme <rule_name> :
    CogitoCapture {
        // Preconditions
        <condition_1> &&
        <condition_2> &&
        ...
    }
    CogitoAction {
        // Actions to be executed when triggered
        <command_1>;
        <command_2>;
        ...
        // Some useful built-in functions here could be :
        // CogitoQuit(): Stop processing cogito rules.
        // CogitoRetry(): Restart processing cogito rules.
    }
)
```

Respondo: From *latin*, to answer

- It corresponds to the motor part of the use of language.
- It can be used to split what system wants to say in some individual parts easier to generate.



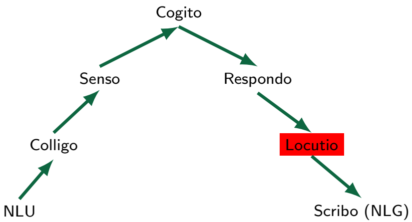
Respondo: From *latin*, to answer

Respondo rules template: RespondoSchemata section

```
(RespondoScheme <rule_name> : [ output ]
    RespondoCapture {
        // Preconditions
        <condition_1> &&
        <condition_2> &&
        ...
    }
    RespondoAction {
        // Actions to be executed when triggered
        <command_1>;
        <command_2>;
        ...
        // ^OBJRESPONDO stands for output expression.
    }
)
```

Locutio: From *latin*, to speak

- It may be used to sort expressions given by `Respondo` module in order to make the dialogue more natural.
- For example, if we want to say lots of thing, some of them (for example questions) should be put at the end of the dialogue act.



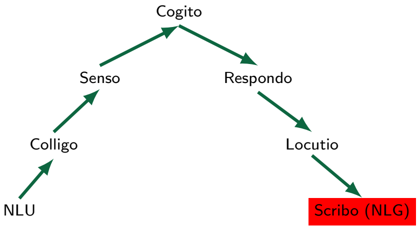
Locutio: From *latin*, to speak

Locutio rules template: LocutioSchemata section

```
(LocutioScheme <rule_name> : [ in_1 ... in_n >> output ]
   LocutioCapture {
      // Preconditions
      <condition_1> &&
      <condition_2> &&
      ...
   }
   LocutioAction {
      // Actions to be executed when triggered
      <command_1>;
      <command_2>;
      ...
      // ^OBJLOCUTIO stands for output expression.
      // #OBJRESPONDO-1 stands for first input term.
      // #OBJRESPONDO-2 stands for second input term.
      // #OBJRESPONDO-N stands for nth input term.
   }
)
```

Scribo: From *latin*, to write

- It's used to generate system proferences into target natural language.
- So it's a language dependant module (as well as lexicon and grammar ones).
- Related section in main project file: `scriboModel forLanguage <...>`.



Scribo: From *latin*, to write

Scribo rules template: ScriboSchemata subsection

```
(ScriboScheme <rule_name> : [ in_1 ... in_n ]
   ScriboCapture {
      // Preconditions
      <condition_1> &&
      <condition_2> &&
      ...
   }
   ScriboAction {
      // Actions to be executed when triggered
      <command_1>;
      <command_2>;
      ...
      // #OBJLOCUTIO-1 stands for first input term.
      // #OBJLOCUTIO-2 stands for second input term.
      // #OBJLOCUTIO-N stands for nth input term.
   }
)
```

Scribo: From *latin*, to write

Two useful built-in functions here:

- `SetMainAnswerString(string s)`: Generates system answer in accumulative way.
- `SetMainAnswerStringRandom(string s1, string s2, ...)`: Chooses an answer randomly between its arguments.

Example

```
string greet;
cond {
   (ClockAskHour() < 7)  greet <- 'Good night. ';
   (ClockAskHour() < 14) greet <- 'Good morning. ';
   (ClockAskHour() < 20) greet <- 'Good afternoon. ';
   default               greet <- 'Good night. ';
}
SetMainAnswerString( greet );
SetMainAnswerStringRandom(
      'Welcome to the Personal Assistant Service. ',
      'Thanks for contacting the Personal Assistant Service. ',
      'Your Personal Assistant Service speaking. ',
      'Thank you for calling the Personal Assistant Service. ');
```

Session 04: Exercise 01 Dialogue system: Integer calculator

Metatype StructureBatch

- Used to create sequences of some type.
- They are similar to dequeues (you can insert and remove elements in both ends). By the way, positions start in 1.
- But you can also "read" and "write" elements in other positions.

Example

```
classDef:ElementInt( integer )
classDef:StructureBatch( Counters: ( integer ) )
...
Counters counters;
Counter c1 <- 1;
Counter c2 <- 2;
Counter c3 <- 3;

BatchInsertEnd(counters, c1);
BatchInsertEnd(counters, c2);
BatchInsertEnd(counters, c3); // {c1, c2, c3}
```

Metatype StructureBatch

Batch built-in functions

```
integer BatchSize(batch b);
batch BatchJoin(batch b1, batch b2);

procedure BatchExtractInit(batch b, elem out);
procedure BatchExtractEnd(batch b, elem out);
procedure BatchInsertInit(batch, elem in);
procedure BatchInsertEnd(batch b, elem in);

procedure BatchRecoverPosition
          (batch b, integer pos, elem out);
procedure BatchAssignPosition
          (batch b, integer pos, elem in);

procedure BatchExchange
          (batch b, integer pos1, integer pos2);
```

Session 04: Exercise 02
Dialogue system: Basic domotic assistant