

# 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:

```
1 lektaProject
2
3   projectHead
4     <...>
5
6   projectSetup
7     <...>
8
9   classModel
10    <...>
11
12   lexicalModel forLanguage <...>
13     <...>
14
15   grammaticalModel forLanguage <...>
16     <...>
```



# 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:

```
1 lektaProject
2
3   projectHead
4     <...>
5
6   projectSetup
7     <...>
8
9   classModel
10    <...>
11
12   lexicalModel forLanguage <...>
13     <...>
14
15   grammaticalModel forLanguage <...>
16     <...>
```

# 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:

```
1 lektaProject
2
3     projectHead
4         <...>
5
6     projectSetup
7         <...>
8
9     classModel
10        <...>
11
12    lexicalModel forLanguage <...>
13        <...>
14
15    grammaticalModel forLanguage <...>
16        <...>
```

# 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.

```
1 projectHead
2   // Languages defined for this project.
3   projectLanguageScope : [ anbm ]
4
5   // Output file after compiling.
6   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.

```
1 projectHead
2   // Languages defined for this project.
3   projectLanguageScope : [ anbm ]
4
5   // Output file after compiling.
6   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.

```
1 projectHead
2   // Languages defined for this project.
3   projectLanguageScope : [ anbm ]
4
5   // Output file after compiling.
6   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.**

```
1 projectHead
2   // Languages defined for this project.
3   projectLanguageScope : [ anbm ]
4
5   // Output file after compiling.
6   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.**

```
1 projectHead
2   // Languages defined for this project.
3   projectLanguageScope : [ anbm ]
4
5   // Output file after compiling.
6   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**.

```
1 projectHead
2   // Languages defined for this project.
3   projectLanguageScope : [ anbm ]
4
5   // Output file after compiling.
6   projectCompileOutput : ".AnBm.olk"
```



## AnBm.lkt (2/4)

```
1 projectSetup
2   // Possible output parser types.
3   setupParserRoots = S
4
5 classModel
6   // Definition of all the types needed.
7   // Type Void acts as a label.
8   classDef:Void ( S, A, B, a, b )
9
10  // Lexical model is language dependant
11  lexicalModel forLanguage anbm
12    // Lexicon elements that we must detect.
13    // Here a and b act as grammar terminal
14    // symbols.
15    ("a", a)
16    ("b", b)
```

# AnBm.lkt (3/4)

```
1 // Grammatical model is language dependant
2 grammaticalModel forLanguage anbmn
3 // List of grammar rules
4 /* 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
   */
5 (R1: [ S -> a A b B ])
6 (R2: [ A -> ])
7 (R3: [ A -> a A ])
8 (R4: [ B -> ])
9 (R5: [ B -> b B ])
```

# AnBm.lkt (4/4)

```

1 // *****
2 //
3 // Exercise 01: Generator/Recognizer for language AnBm. Where n,m >= 1
4 //
5 // *****
6
7 lektaProject
8   projectHead
9     projectLanguageScope : [ anbm ]
10    projectCompileOutput : ".AnBm.olk"
11
12   projectSetup
13     setupParserRoots = S
14
15   classModel
16     classDef:Void ( S, A, B, a, b )
17
18   lexicalModel forLanguage anbm
19     ("a", a)
20     ("b", b)
21
22   grammaticalModel forLanguage anbm
23     (R1: [ S -> a A b B ])
24     (R2: [ A -> ])
25     (R3: [ A -> a A ])
26     (R4: [ B -> ])
27     (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`

```
1 // Start lekta engine
2 LaunchLektaKernel()
3
4 // Use recently compiled project
5 UseProject (ProjectCompile : ".AnBm.olk")
6
7 // Options for visualization
8 DisplayProcessUnderstandingOn
9
10 // Start a dialogue with lekta
11 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`

```
1 // Start lekta engine
2 LaunchLektaKernel()
3
4 // Use recently compiled project
5 UseProject (ProjectCompile : ".AnBm.olk")
6
7 // Options for visualization
8 DisplayProcessUnderstandingOn
9
10 // Start a dialogue with lekta
11 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`

```
1 // Start lekta engine
2 LaunchLektaKernel()
3
4 // Use recently compiled project
5 UseProject (ProjectCompile : ".AnBm.olk")
6
7 // Options for visualization
8 DisplayProcessUnderstandingOn
9
10 // Start a dialogue with lekta
11 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`

```
1 // Start lekta engine
2 LaunchLektaKernel()
3
4 // Use recently compiled project
5 UseProject (ProjectCompile : ".AnBm.olk")
6
7 // Options for visualization
8 DisplayProcessUnderstandingOn
9
10 // Start a dialogue with lekta
11 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`

```
1 // Start lekta engine
2 LaunchLektaKernel()
3
4 // Use recently compiled project
5 UseProject (ProjectCompile : ".AnBm.olk")
6
7 // Options for visualization
8 DisplayProcessUnderstandingOn
9
10 // Start a dialogue with lekta
11 CreateDialogue()
```



# Session 01: Exercise 01

# Grammar rules syntax

```
1 (<rule_label>:  
2   [ <symbol> -> <list_of_symbols> ]  
3   {  
4       ...  
5       <commands>  
6       ...  
7   }  
8 )
```

- <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

```
1 (<rule_label>:  
2   [ <symbol> -> <list_of_symbols> ]  
3   {  
4       ...  
5       <commands>  
6       ...  
7   }  
8 )
```

- <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'.

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

```
1 // These rules are expanded into standard rules  
2 (R1: [ S -> A B ])  
3 (R1: [ S -> B ])  
4 (R1: [ S -> A ])  
5 (R1: [ S -> ])  
6  
7 (R2: [ A -> a A ])  
8 (R2: [ A -> a ])
```

## Special features: Optional symbol (?)

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

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

```
1 // These rules are expanded into standard rules  
2 (R1: [ S -> A B ])  
3 (R1: [ S -> B ])  
4 (R1: [ S -> A ])  
5 (R1: [ S -> ])  
6  
7 (R2: [ A -> a A ])  
8 (R2: [ A -> a ])
```

## Special features: Or symbol (|)

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

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

```
1 // These rules are expanded into standard rules
```

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

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

```
4 (R1: [ S -> A D ])
```

Special symbols can be combined:

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

```
2
```

```
3 (R1: [ S -> A ])
```

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

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

## Special features: Or symbol (|)

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

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

```
1 // These rules are expanded into standard rules
```

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

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

```
4 (R1: [ S -> A D ])
```

Special symbols can be combined:

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

```
2
```

```
3 (R1: [ S -> A ])
```

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

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

## Special features: Or symbol (|)

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

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

```
1 // These rules are expanded into standard rules
```

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

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

```
4 (R1: [ S -> A D ])
```

Special symbols can be combined:

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

```
2
```

```
3 (R1: [ S -> A ])
```

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

```
5 (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.

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

```
1 // These rules are expanded into standard rules
```

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

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

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

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

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

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

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

```
1 // 12 Rules when expanding
```

```
2 (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.

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

```
1 // These rules are expanded into standard rules
```

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

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

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

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

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

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

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

```
1 // 12 Rules when expanding
```

```
2 (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.

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

```
1 // These rules are expanded into standard rules
```

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

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

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

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

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

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

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

```
1 // 12 Rules when expanding
```

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

## Special features: Left and right limits ( $\&[+^{\wedge}]$ and $[+^{\$}]\&$ )

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

```
1 (R1: [ S ->  $\&[+^{\wedge}]$  A B ])
```

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

```
1 (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:

```
1 (R1: [ S ->  $\&[+^{\wedge}]$  A B  $[+^{\$}]\&$  ])
```

## Special features: Left and right limits ( $\&[+^{\wedge}]$ and $[+ \$]\&$ )

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

```
1 (R1: [ S ->  $\&[+^{\wedge}]$  A B ])
```

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

```
1 (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:

```
1 (R1: [ S ->  $\&[+^{\wedge}]$  A B  $[+ \$]\&$  ])
```

## Special features: Left and right limits ( $\&[+^{\wedge}]$ and $[+^{\$}]\&$ )

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

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

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

```
1 (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:

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

# Session 01: Exercise 02

# Session 01: Exercise 03



# 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

```
1 ...  
2 ElementInt i <- 5;  
3 ...
```

## Correct Example

```
1 classDef:ElementInt ( integer )  
2 ...  
3 integer i <- 5;  
4 ...
```

# 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

```
1 ...  
2 ElementInt i <- 5;  
3 ...
```

## Correct Example

```
1 classDef:ElementInt ( integer )  
2 ...  
3 integer i <- 5;  
4 ...
```

## 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

```
1 ...  
2 ElementInt i <- 5;  
3 ...
```

### Correct Example

```
1 classDef:ElementInt ( integer )  
2 ...  
3 integer i <- 5;  
4 ...
```

# 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

```
1 ...  
2 ElementInt i <- 5;  
3 ...
```

## Correct Example

```
1 classDef:ElementInt ( integer )  
2 ...  
3 integer i <- 5;  
4 ...
```

# Type specification in Lekta

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

```
1 classDef:ElementInt      ( int )
2 classDef:ElementBool    ( boolean )
3 classDef:ElementLiteral ( String )
4 ...
5 int i <- 5;
6 boolean flag <- False;
7 String s <- 'this is a string';
8 ...
```



# Type specification in Lekta

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

```
1 classDef:ElementInt      ( int )
2 classDef:ElementBool    ( boolean )
3 classDef:ElementLiteral ( String )
4 ...
5 int i <- 5;
6 boolean flag <- False;
7 String s <- 'this is a string';
8 ...
```

# Complex structures

```
1 classDef:ElementInt      (Counter)
2 classDef:ElementBool    (Flag)
3 classDef:ElementLiteral (Expression)
4 classDef:StructureComplex
5 (
6     ExampleStructure:
7     (
8         Counter,
9         Flag,
10        Expression
11    )
12 )
13 ...
14 ExampleStructure a;
15 a.Counter <- 5;
16 a.Flag <- True;
17 a.Expression <- 'this is a string';
18 ...
```

- Previous exercises only make some syntactic analysis of the language.
- So we can only recognize valid sentences in that language (parser 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.

```
1 classDef:ElementInt ( counterN, counterM )
2 classDef:StructureComplex ( S:( counterN, counterM ) )
3 classDef:StructureComplex ( A:( counterN ) )
4 classDef:StructureComplex ( B:( counterM ) )
5 classDef:Void ( a, b )
```

- Previous exercises only make some syntactic analysis of the language.
- So we can only recognize valid sentences in that language (parser 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.

```
1 classDef:ElementInt ( counterN, counterM )  
2 classDef:StructureComplex ( S:( counterN, counterM ) )  
3 classDef:StructureComplex ( A:( counterN ) )  
4 classDef:StructureComplex ( B:( counterM ) )  
5 classDef:Void ( a, b )
```

- Previous exercises only make some syntactic analysis of the language.
- So we can only recognize valid sentences in that language (parser 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.

```
1 classDef:ElementInt ( counterN, counterM )  
2 classDef:StructureComplex ( S:( counterN, counterM ) )  
3 classDef:StructureComplex ( A:( counterN ) )  
4 classDef:StructureComplex ( B:( counterM ) )  
5 classDef:Void ( a, b )
```

- Previous exercises only make some syntactic analysis of the language.
- So we can only recognize valid sentences in that language (parser 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.

```
1 classDef:ElementInt ( counterN, counterM )  
2 classDef:StructureComplex ( S:( counterN, counterM ) )  
3 classDef:StructureComplex ( A:( counterN ) )  
4 classDef:StructureComplex ( B:( counterM ) )  
5 classDef:Void ( a, b )
```

- Previous exercises only make some syntactic analysis of the language.
- So we can only recognize valid sentences in that language (parser 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.

```
1 classDef:ElementInt ( counterN, counterM )
2 classDef:StructureComplex ( S:( counterN, counterM ) )
3 classDef:StructureComplex ( A:( counterN ) )
4 classDef:StructureComplex ( B:( counterM ) )
5 classDef:Void ( a, b )
```

```
1 (R1: [ S -> A B ] {  
2     ^.counterN <- #1.counterN;  
3     ^.counterM <- #2.counterM; } )  
4  
5 (R2: [ A -> a A ] {  
6     ^.counterN <- 1 + #2.counterN; } )  
7  
8 (R3: [ A -> a ] {  
9     ^.counterN <- 1; } )  
10  
11 (R4: [ B -> b B ] {  
12     ^.counterM <- 1 + #2.counterM; } )  
13  
14 (R5: [ B -> b ] {  
15     ^.counterM <- 1; } )
```

Note special syntax:

- $\wedge$  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

# Session 02: Exercise 01

# Files inclusion

```
1 lektaProject
2
3   projectHead
4       projectLanguageScope : [ en ]
5       projectCompileOutput : ".Numbers.olk"
6
7   projectSetup
8       setupParserRoots = Number
9
10  classModel
11      #Include "NumberTypes.lkt"
12
13  lexicalModel forLanguage en
14      #Include "NumberEnglishLexicon.lkt"
15
16  grammaticalModel forLanguage en
17      #Include "NumberEnglishGrammar.lkt"
```

## Function and procedure declaration

```
1 classDef:ElementInt ( integer )
2 classDef:ElementBool ( bool )
3 // Templates
4 <output_type> function_name(<parameter_list>) {
5     ...
6 }
7 procedure function_name(<parameter_list>) {
8     ...
9 }
10
11 // Examples
12 bool f1(integer i) {
13     ...
14 }
15 procedure f2(integer i) {
16     ...
17 }
18 procedure f3() { // Not "void" keyword
19     ...
20 }
```

## Comments

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

## Arithmetic operators

```
1 a <- b + c; // Addition
2 a <- b - c; // Subtraction
3 a <- b * c; // Multiplication
4 a <- b / c; // Division
5 a++;       // Post-autoincrement
6 ++a;       // Pre-autoincrement
7 a--;       // Post-autodecrement
8 --a;       // Pre-autodecrement
```

## Comments

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

## Arithmetic operators

```
1 a <- b + c; // Addition
2 a <- b - c; // Subtraction
3 a <- b * c; // Multiplication
4 a <- b / c; // Division
5 a++;       // Post-autoincrement
6 ++a;       // Pre-autoincrement
7 a--;       // Post-autodecrement
8 --a;       // Pre-autodecrement
```

## Comparison operators

```
1 a > b // Greater
2 a >= b // Greater or equal
3 a < b // Less
4 a <= b // Less or equal
5 a == b // Equal
6 a != b // Not equal
```

## Boolean operators

```
1 a && b // And
2 a || b // Or
3 !! a // Not
```

## Comparison operators

```
1 a > b // Greater
2 a >= b // Greater or equal
3 a < b // Less
4 a <= b // Less or equal
5 a == b // Equal
6 a != b // Not equal
```

## Boolean operators

```
1 a && b // And
2 a || b // Or
3 !! a // Not
```



# Lazy evaluation

```
1 boolean f1()  
2 {  
3     // Writes a message to standard output  
4     SpyMessage("Message from f1");  
5     return False;  
6 }  
7  
8 boolean f2()  
9 {  
10    SpyMessage("Message from f2");  
11    return True;  
12 }  
13  
14 procedure testingLazyEvaluation()  
15 {  
16     boolean b;  
17  
18     b <- f1() && f2(); // "Message from f1"  
19     b <- f2() || f1(); // "Message from f2"  
20 }
```

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

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

# Programming structures: switch

```
1 switch (month)
2 {
3     case 1 { ret <- 'January' ;}
4     case 2 { ret <- 'February' ;}
5     case 3 { ret <- 'March' ;}
6     case 4 { ret <- 'April' ;}
7     case 5 { ret <- 'May' ;}
8     case 6 { ret <- 'June' ;}
9     case 7 { ret <- 'July' ;}
10    case 8 { ret <- 'August' ;}
11    case 9 { ret <- 'September' ;}
12    case 10 { ret <- 'October' ;}
13    case 11 { ret <- 'November' ;}
14    default { ret <- 'December' ;}
15 }
```

## Programming structures: cond

```
1 cond
2 {
3     (month == 1) { ret <- 'January'; }
4     (month == 2) { ret <- 'February'; }
5     (month == 3) { ret <- 'March'; }
6     (month == 4) { ret <- 'April'; }
7     (month == 5) { ret <- 'May'; }
8     (month == 6) { ret <- 'June'; }
9     (month == 7) { ret <- 'July'; }
10    (month == 8) { ret <- 'August'; }
11    (month == 9) { ret <- 'September'; }
12    (month == 10) { ret <- 'October'; }
13    (month == 11) { ret <- 'November'; }
14    default { ret <- 'December'; }
15 }
```

# Programming structures: loops

## “While” loop

```
1 integer position, size;  
2 ...  
3 position <- 1;  
4 while (position <= size) {  
5     <...>  
6     position++;  
7 }
```

## “For” loop

```
1 integer position, size;  
2 ...  
3 for (position <- 1; position <= size; position++) {  
4     <...>  
5 }
```

# Programming structures: loops

## “While” loop

```
1 integer position, size;  
2 ...  
3 position <- 1;  
4 while (position <= size) {  
5     <...>  
6     position++;  
7 }
```

## “For” loop

```
1 integer position, size;  
2 ...  
3 for (position <- 1; position <= size; position++) {  
4     <...>  
5 }
```

# Built-in functions

## Date & time

```
1 integer ClockAskYear();  
2 integer ClockAskMonth();  
3 integer ClockAskDayOfTheMonth();  
4 integer ClockAskDayOfTheWeek();  
5 integer ClockAskHour();  
6 integer ClockAskMinute();  
7 integer ClockAskSecond();
```

## Atomic types transformations

```
1 bool ShapeToBool();  
2 integer ShapeToInt();  
3 real ShapeToReal();  
4 string ShapeToLiteral(); // Same as ShapeToString();  
5 message ShapeToMessage();  
6 range ShapeToRange();
```

# Built-in functions

## Date & time

```
1 integer ClockAskYear();  
2 integer ClockAskMonth();  
3 integer ClockAskDayOfTheMonth();  
4 integer ClockAskDayOfTheWeek();  
5 integer ClockAskHour();  
6 integer ClockAskMinute();  
7 integer ClockAskSecond();
```

## Atomic types transformations

```
1 bool ShapeToBool();  
2 integer ShapeToInt();  
3 real ShapeToReal();  
4 string ShapeToLiteral(); // Same as ShapeToString();  
5 message ShapeToMessage();  
6 range ShapeToRange();
```



# Built-in functions

## Literal functions

```
1 string LiteralConvertLower(string in);
2 string LiteralConvertUpper(string in);
3 string LiteralConcat(string in1, string in2);
4 string LiteralSubstitution
5     (string in, string from, string to);
6 string LiteralGlobalSubstitution
7     (string in, string from, string to);
8 integer LiteralSize(string in);
9 string LiteralPositionValue(string in, integer pos);
10 string LiteralSearch(string in, string toLookFor);
11 bool LiteralIncluded(string toLookFor, string in);
12 string SubLiteral
13     (string in, integer from, integer to);
```

# Built-in functions

## “Filled” and “Devoid”

```
1 classDef:ElementInt( f1, f2 )
2 classDef:StructureComplex( F: (f1, f2) )
3 ...
4
5 F f;
6 Filled(f);      // False
7 Devoid(f);      // True
8
9 f.f1 <- 5;
10 Filled(f);      // True
11 Devoid(f);      // False
12 Filled(f.f1);   // True
13 Devoid(f.f1);   // False
14 Filled(f.f2);   // False
15 Devoid(f.f2);   // True
16
17 if(f)          { <...> } // Same as Filled(f)
18 if(!! f)       { <...> } // Same as Devoid(f)
```

# Lexicon

```
1 classDef:Void( one, two )
2 classDef:ElementInt( NumberValue )
3 classDef:StructureComplex( Number: (NumberValue) )
4
5 ...
6
7 ("one", one)
8 ("two", two)
9
10 ("one", NumberValue, 1)
11 ("two", NumberValue, 2)
12
13 ("one", Number, (NumberValue: 1))
14 ("two", Number, (NumberValue: 2))
```

# Session 02: Exercise 02

# Assignment operator

```
1 classDef:ElementInt( f1, f2, f3, f4 )
2 classDef:StructureComplex( S: (f1, f2, f3, f4) )
3
4 ...
5
6 S a;
7 a.f2 <- 2;
8 a.f4 <- 4;
9
10 S b;
11 b.f3 <- 3;
12 b.f4 <- 4;
```

a:  $\begin{bmatrix} \text{f2:} & 2 \\ \text{f4:} & 4 \end{bmatrix}$

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

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

a <- b;

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

# Overwrite operator

```
1 classDef:ElementInt( f1, f2, f3, f4 )
2 classDef:StructureComplex( S: (f1, f2, f3, f4) )
3
4 ...
5
6 S a;
7 a.f2 <- 2;
8 a.f4 <- 4;
9
10 S b;
11 b.f3 <- 3;
12 b.f4 <- 4;
```

a:  $\begin{bmatrix} \text{f2:} & 2 \\ \text{f4:} & 4 \end{bmatrix}$

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

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

a <| b;

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