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

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

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. `ElementRange`: Enumerate type.
  6. `StructureBatch`: Sequence of elements of the same type.
  7. `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. `ElementRange`: Enumerate type.
  6. `StructureBatch`: Sequence of elements of the same type.
  7. `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. `ElementRange`: Enumerate type.
  6. `StructureBatch`: Sequence of elements of the same type.
  7. `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

Session 02: Exercise 01

Function and procedure declaration

```
classDef:ElementInt ( integer )
classDef:ElementBool ( bool )

<ouput_type> function_name(<parameter_list>) {
    ...
}

bool f1(integer i) {
    ...
}

procedure function_name(<parameter_list>) {
    ...
}

procedure f2(integer i) {
    ...
}

EMPTY PARAMETER LIST
```

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

Output

Lazy evaluation

Output

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: while

```
// TODO
```

Programming structures: for

```
// TODO
```