TNO ESI

Powered by industry and academia

DOMAIN-SPECIFIC LANGUAGES (DSL)

Software Systems (Computer & Embedded Systems Engineering)

Rosilde Corvino

14 – 01 – 2025

AGENDA OF THE COURSE

	Week 6 (17-12)	Week 6 (19-12)	Week 7 (7-1)	Week 7 (9-1)	Week 8 (14-1)	Week 8 (16 -1)	Week 9 (21-1)	Week 10
Lectures on Tuesdays (2 hours)	Introduction		StateChart 1		DSL 1			
	UML 1		StateChart 2		DSL 2			
Labs on Thursdays (4 hours)		1 UML Lect 3 labs		StateChart		**4 hours lab**		
Assignment due on Friday				UML		Statechart		DSL + Reflection

QUIZ GAMES

- Identified by the banner: **Think/Write → Pair → Share**

- Instructions:
  1. Divide into teams of two students
  2. Discuss the solution to the quiz together
  3. Volunteer or be asked to share
  4. Points will be awarded for participation:
     1. Every time you share during a game, you earn 0.3 points
     2. You can earn up to a maximum of 1 additional point on the final note for this part of the course
     3. Do not forget to write your name on the winners' sheet after the lecture

OBJECTIVES

- At the end of the course, you should be able to:
  - Explain the purpose of Domain-Specific Languages
  - Explain the basics of formal grammar and parsing
  - Create basic textual Domain-Specific Languages and review examples of validation and generators

- Assessment:
  - Modeling assignment using Domain-Specific Languages (in groups of 2 students)
  - Reflection document on Model-Based Development (individual)

AGENDA FOR DOMAIN-SPECIFIC LANGUAGE

- 15 minutes Introduction
- 30 minutes Formal grammar and Parsing
- 15 minutes Break
- 30 minutes How to design a DSL and its grammar
- 10 minutes Lark parser generator and Transformer and Validator
- 15 minutes General conclusions

INTRODUCTION

WHAT DO YOU ALREADY KNOW?

1. What do you associate with the term Domain-Specific Language?

2. Do you know any Domain-Specific Languages?

Duration: 1 minute of discussion with your partner and then speak up

Think → Pair → Share

A DSL CAN BE ASSOCIATED WITH A JARGON. WHAT IS JARGON?

<u>Oxford dictionaries:</u>

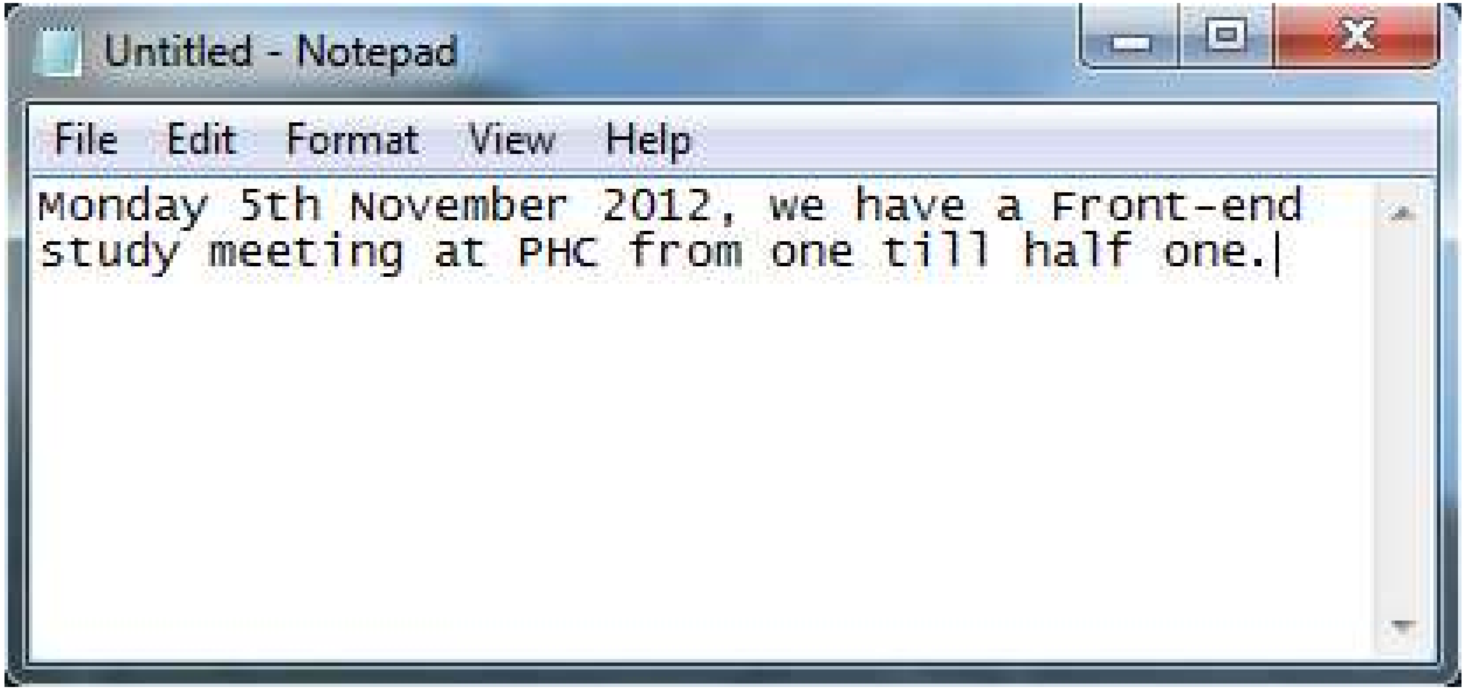
- Special words or expressions used by a profession or group
- that are difficult for others to understand

<u>Wikipedia:</u>

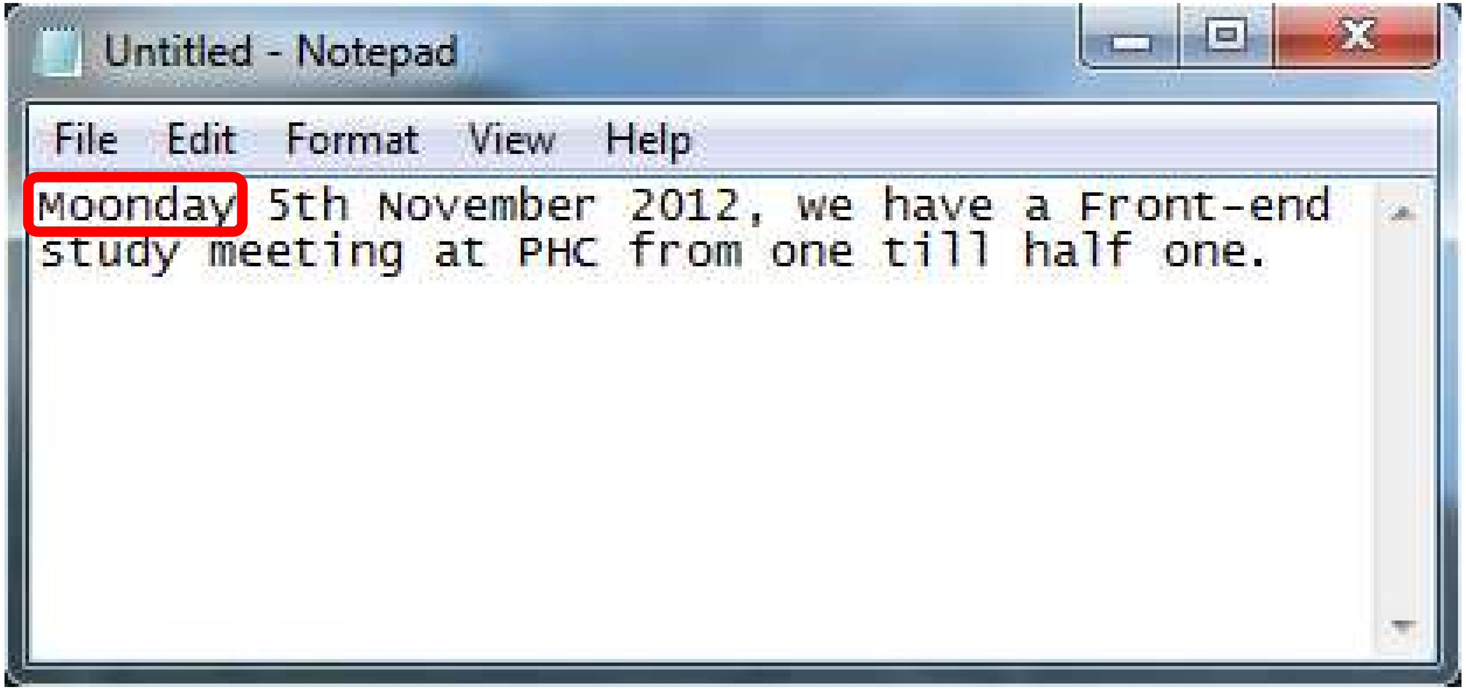
- Terminology defined in relationship to a specific activity, profession, group, or event
- … a barrier to communication with those not familiar with the language

➢ A standard term may be given a more precise or unique usage

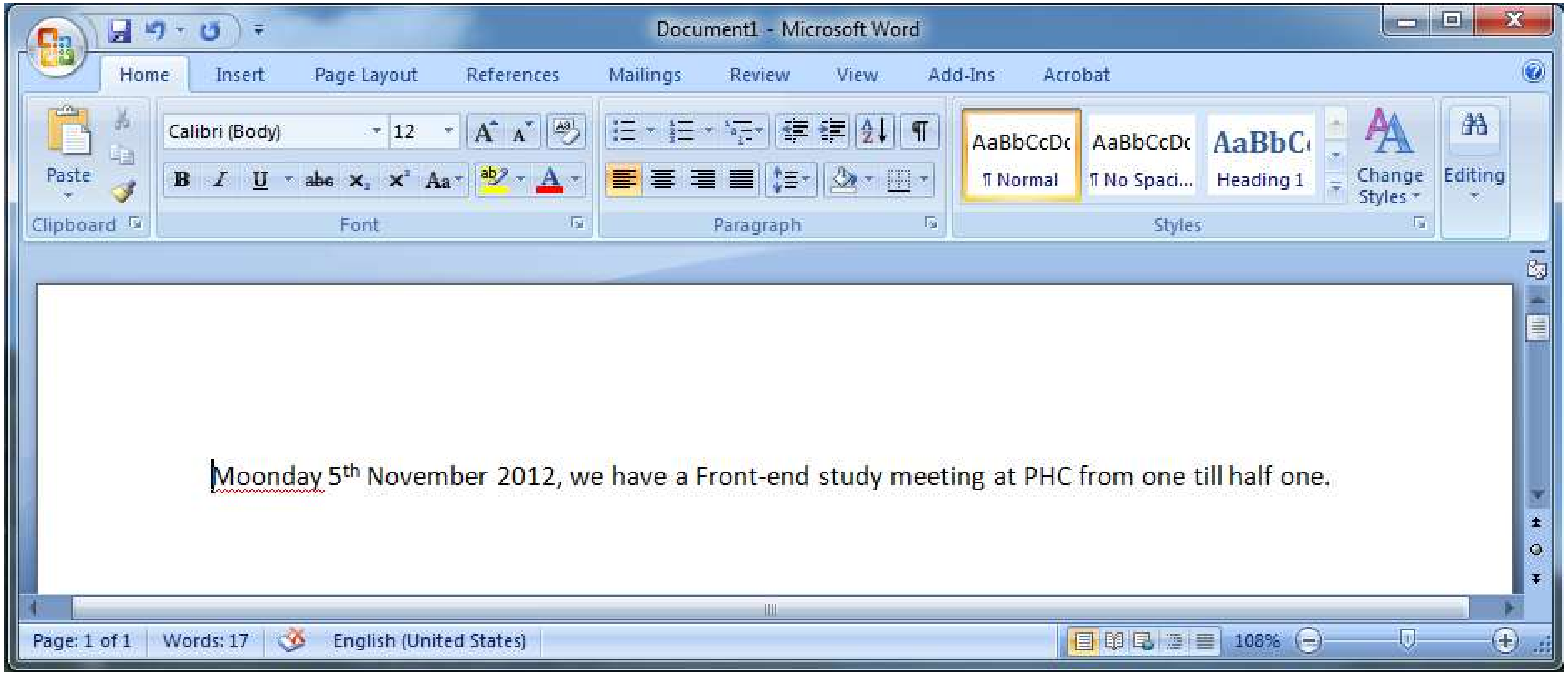
MICROSOFT NOTEPAD: A CORRECT SENTENCE



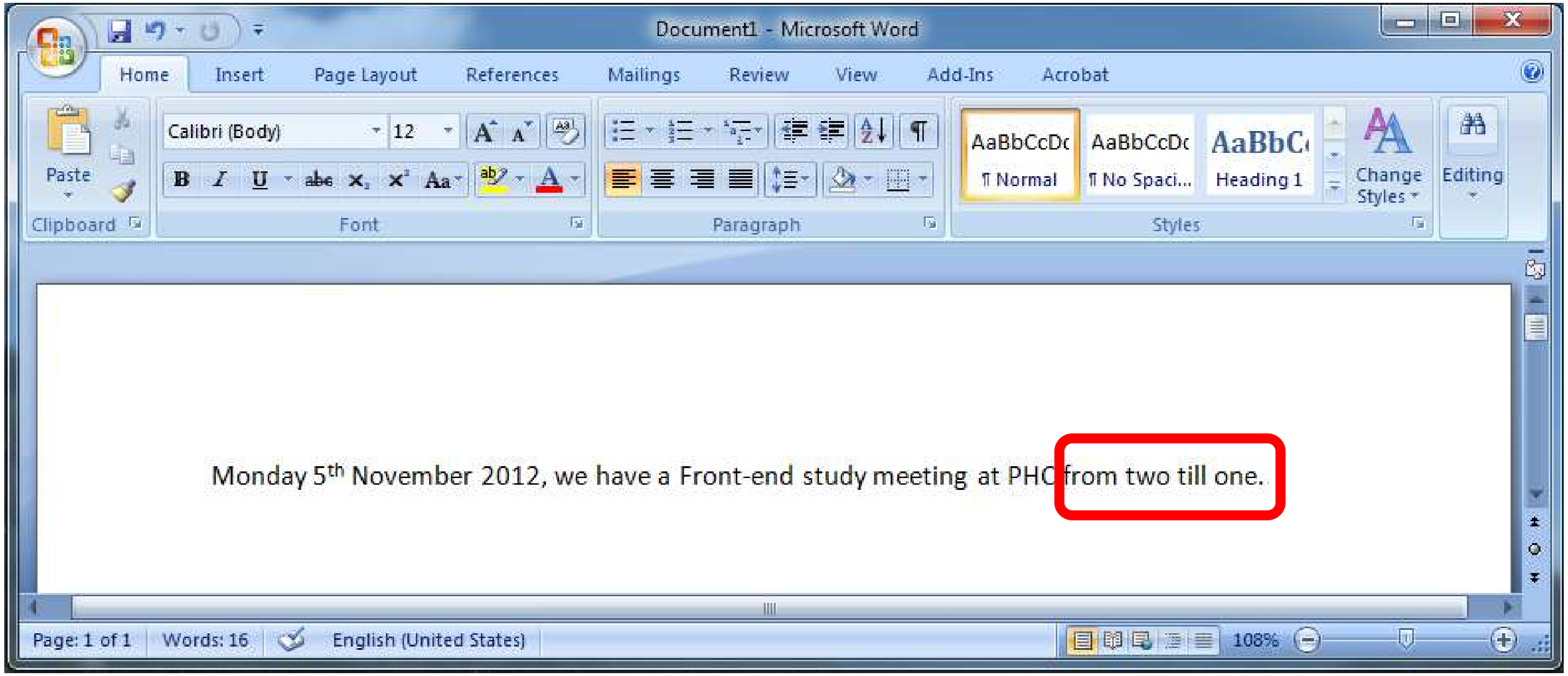
MICROSOFT NOTEPAD: NO PROBLEM DETECTED, BUT …



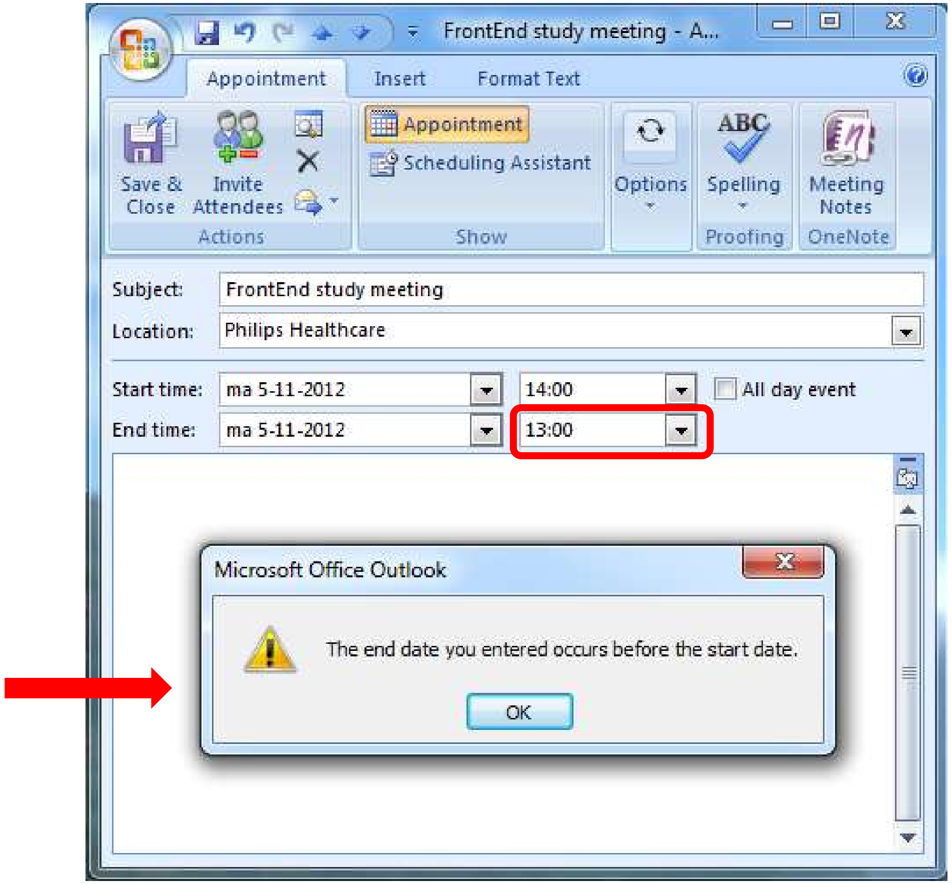
MICROSOFT WORD: SPELLING ERROR



MICROSOFT WORD: NO PROBLEM DETECTED, BUT …



MICROSOFT OUTLOOK: WRONG TIMES



DO YOU KNOW ANY DOMAIN-SPECIFIC LANGUAGES?

• PlantUML	for	Unified Modeling Language (UML)
• CREATE Statechart Tools	for	Finite-State Machines (FSM)

THE SCOPE OF A LANGUAGES

Universal, across domains

- **General-purpose** programming languages:
  - C, C++, Java, Python, etc.

```c
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

Specific, across diverse contexts in a domain

- **Horizontal** Domain-specific languages:
  - HTML for web pages
  - SQL for relational database queries

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Hello HTML</title>
  </head>
  <body>
    <p>Hello World!</p>
  </body>
</html>
```

```sql
SELECT *
 FROM Book
 WHERE price > 100
 ORDER BY title;
```

Specific to a project or a context within a domain

- **Vertical** Domain-specific languages:
  - Designed for a specific application by a single company

ABSTRACTION LEVELS OF PROGRAMMING LANGUAGES

Abstraction semantics

Metaprogramming (manipulate, generate, or reason about code itself)

Specific tasks and workflows

Domain specific languages

```
SELECT *
 FROM  Book
 WHERE price > 100.00
 ORDER BY title;
```

Algorithm and data structures

High-level programming languages (Python, C++, Java, ..)

```
if (frameSize.height > screenSize.height) {
    frameSize.height = screenSize.height;
}
if (frameSize.width > screenSize.width) {
    frameSize.width = screenSize.width;
}
```

Machine Instructions

Assembly language

```
add     eax, edx
shl     eax, 2
add     eax, edx
shr     eax, 8
```

Hardware operations

Machine code

op	rs	rt	address
100011	00011	01000	00000 00001 000100

tool generators

code generator

compiler

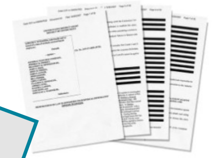
assembler

DSL AS A SINGLE SOURCE OF TRUTH

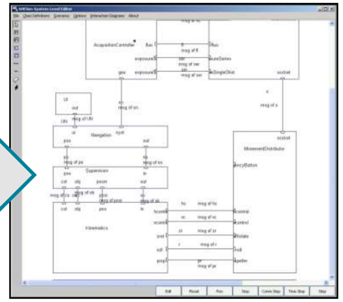
```
restriction VeryCloseTableTopAndBeam
    activation
           Distance(TableTop, Beam) < 20 mm
    effects
        userGuidance "TableTop and Beam very close"
        relative limit TableTop*[Rotation, Translation],
                       Beam*[Rotation, Translation]
            at 0


restriction ApproachingTableTopAndBeam
    activation
           Distance(TableTop, Beam) < 35 mm + 15 cm
    effects
        userGuidance "TableTop and Beam approaching"
        relative limit TableTop*[Rotation, Translation],
                       Beam*[Rotation, Translation]
            at (Distance(TableTop, Beam) - 35 mm) / 15 cm
```

Validation

Generate documentation



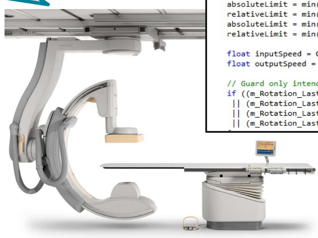
Generate analysis models



Generate source code

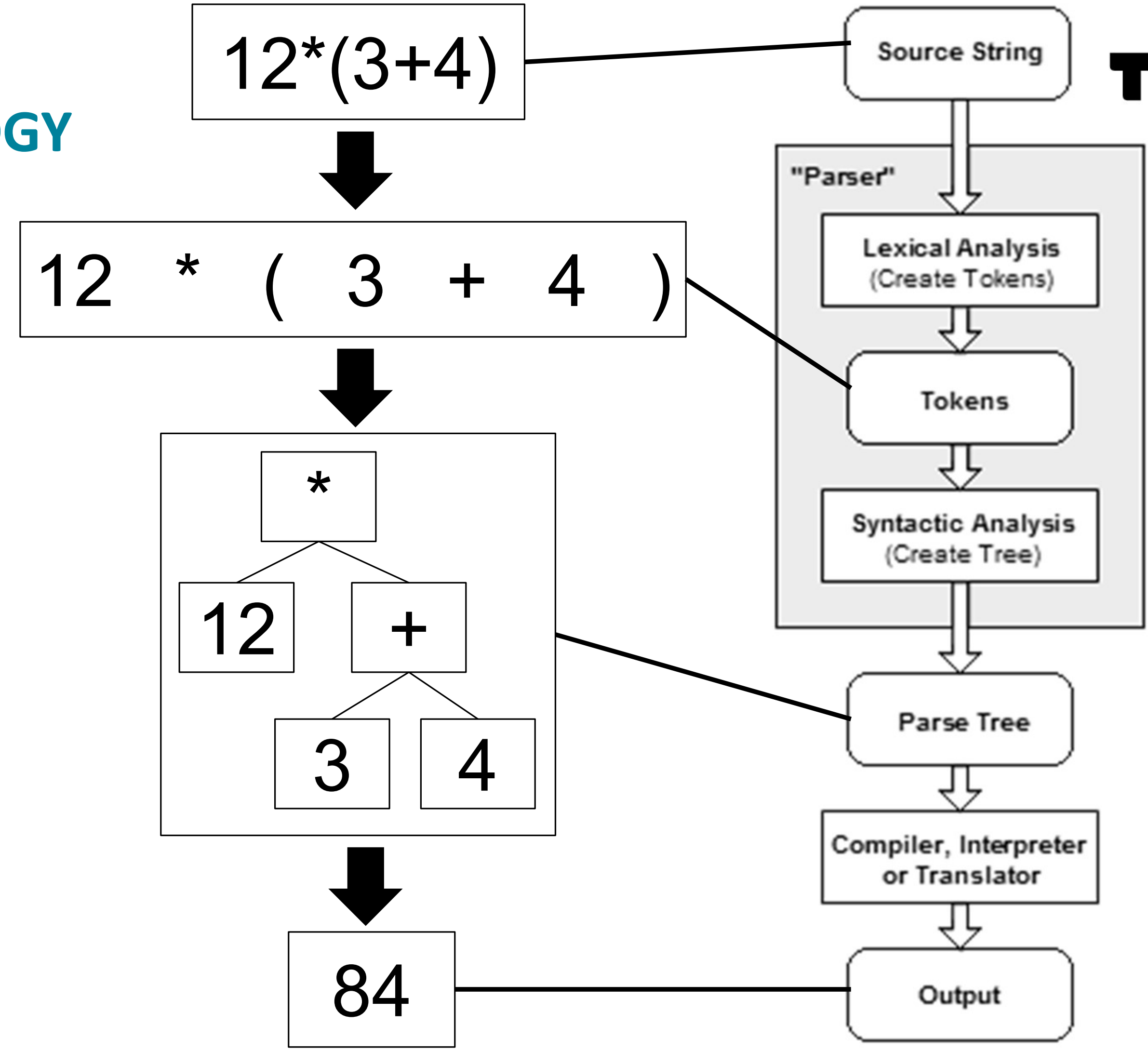
```
float CObject_Beam::GetRotationScaling()
{
    float absoluteLimit = GEN_t_float_max;
    float relativeLimit = 1.;

    absoluteLimit = min(absoluteLimit, m_Rotation_AbsoluteLimit_DetectorRotationInBetween);
    absoluteLimit = min(absoluteLimit, m_Rotation_AbsoluteLimit_VeryCloseTableTopAndBeam);
    relativeLimit = min(relativeLimit, m_Rotation_RelativeLimit_ApproachingTableTopAndBeam);
    absoluteLimit = min(absoluteLimit, m_Rotation_AbsoluteLimit_VeryCloseTableTopAndDetector);
    relativeLimit = min(relativeLimit, m_Rotation_RelativeLimit_ApproachingTableTopAndDetector);
    absoluteLimit = min(absoluteLimit, m_Rotation_AbsoluteLimit_VeryCloseTableBaseAndBeam);
    relativeLimit = min(relativeLimit, m_Rotation_RelativeLimit_ApproachingTableBaseAndBeam);
    absoluteLimit = min(absoluteLimit, m_Rotation_AbsoluteLimit_VeryCloseTableBaseAndDetector);
    relativeLimit = min(relativeLimit, m_Rotation_RelativeLimit_ApproachingTableBaseAndDetector);

    float inputSpeed = GetRequestedRotSpeed();
    float outputSpeed = min(inputSpeed * relativeLimit, absoluteLimit);

    // Guard only intended to reduce calls to logging
    if ((m_Rotation_LastAbsoluteLimit != absoluteLimit)
     || (m_Rotation_LastRelativeLimit != relativeLimit)
     || (m_Rotation_LastInputSpeed != inputSpeed)
     || (m_Rotation_LastOutputSpeed != outputSpeed))
```

factor >30

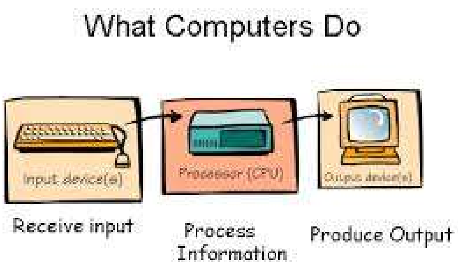


METAMODELS AND GRAMMARS

COMPILER TECHNOLOGY



grammar



EXTENDED BACKUS-NAUR FORM (OR INSPIRED FROM IT)

G = (S, N, T, P)

S		Start symbol	Root of the grammar
N	-	Non-terminals	Finite set of symbols on the LHS of a production rule
T	-	Terminals	Alphabet of the language
P	-	Production rule	LHS can be replaced with RHS

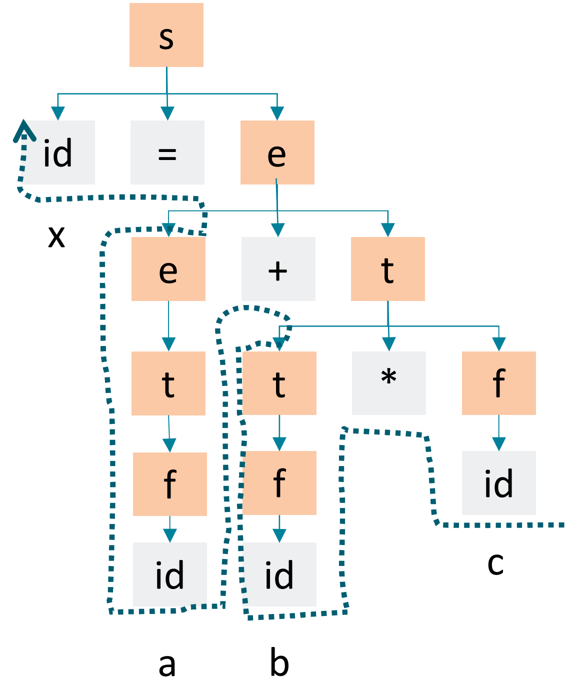
a=b+c

```
start : ID "=" expr

expr  : ID "+" ID

ID    : /[a-zA-Z]+/
```

HOW DOES A PARSER WORK?

line				
1.	s	:	ID "=" e	
2.	e	:	e "+" t	| t
3.	t	:	t "*" f	| f
4.	f	:	ID	

x = a + b * c



7 = 1 + (3 * 2)

KEY CONCEPTS OF EBNF SYNTAX IN LARK

- Repetition: * or +
- Optionality: ?
- Alternatives: |
- Grouping: ( )
- Predefined Data Types
- Aliases: ->

```
grammar = """
start : client_list
client_list : client ("," client)*
client : name ":" location phone_number?
location : "Amsterdam" | "Delft"
name : CNAME
phone_number : NUMBER
%import common.CNAME
%import common.NUMBER
%import common.WS
%ignore WS
"""
```

WHAT IS A VALID PROGRAM ACCORDING TO THIS GRAMMAR?

```
grammar = """
start : client_list
client_list : client ("," client)*
client : name ":" location phone_number?
location : "Amsterdam" | "Delft"
name : CNAME
phone_number : NUMBER
%import common.CNAME
%import common.NUMBER
%import common.WS
%ignore WS
"""
```

Duration: 3-minute discussion with your partner and then speak up

Think → Pair → Share

WHAT IS A VALID PROGRAM ACCORDING TO THIS GRAMMAR?

```
grammar = """
start : client_list
client_list : client ("," client)*
client : name ":" location phone_number?
location : "Amsterdam" -> amsterdam
         | "Delft“ -> delft
name : CNAME
phone_number : NUMBER
%import common.CNAME
%import common.NUMBER
%import common.WS
%ignore WS
"""
```

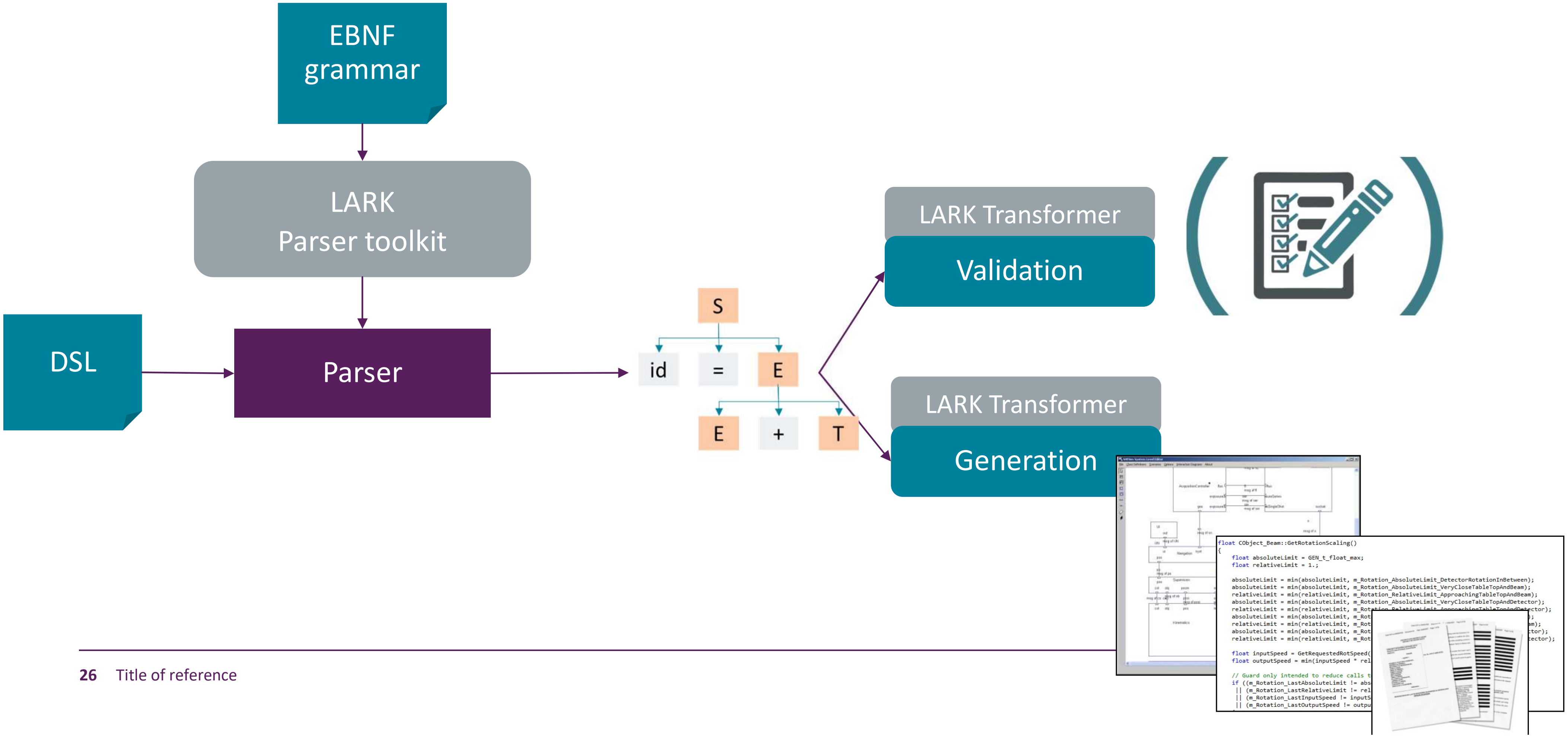
```
program = """
    Alice: Amsterdam 0621445680,
    Bob: Delft 0621445681,
    Charlie: Amsterdam,
    David: Delft
    """
```

Pretty printing

```
start
  client_list
    client
      name        Alice
      location
      phone_number        0621445680
    client
      name        Bob
      location
      phone_number        0621445681
    client
      name        Charlie
      location
    client
      name        David
      location
```

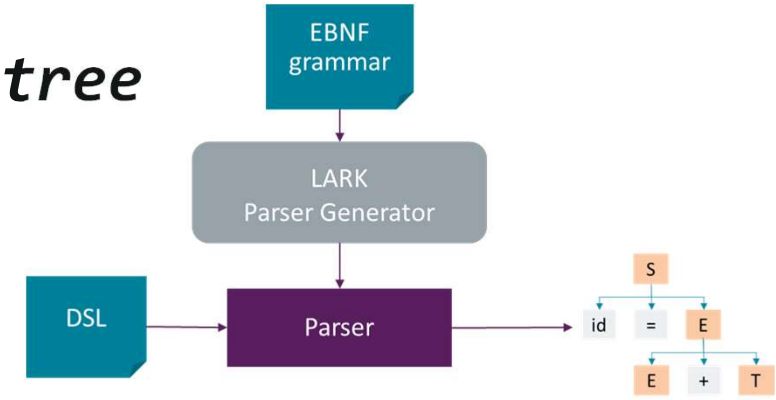
HOW DOES PARSER GENERATION WORK IN LARK?

Written by the user
Generated
Library



HOW IS THE GRAMMAR USED IN LARK?

```
grammar = …
program = …

#define the parser
parser = Lark(grammar, start='start', parser='lalr')

#parse the input program into a tree
tree = parser.parse(program)

#print the tree
print(tree.pretty())
```



LET’S BUILD A DSL AND ITS GRAMMAR

HOW TO DESIGN A DSL?

- Define the problem domain, scope and aim of the DSL: why do we design a DSL?
- Give a few examples of the DSL: what do the examples do?
- Design the DSL syntax: what makes the DSL syntax good?
- Design the DSL grammar: what makes a grammar good?

Example-driven explanation

A language for home security systems configuration

PROBLEM DOMAIN AND SCOPE: WHY DO WE DESIGN A DSL?

- Example: **A language for home security systems configuration**
  - We want to build a DSL to model and simulate a home security system
  - We want to be able to describe various system configurations
    - Th components: cameras and sensors located in different parts of the house
    - Behavior of the system when an intrusion is detected in each location
  - We want to validate the system model
  - From a model, we want to generate parameters to configure a (simple) system simulator

AN EXAMPLE OF A CONFIGURATION MODEL

```
program = """
SYSTEM {
    CAMERAS { living_room  }
    SENSORS { living_room bedroom }
}
SYSTEM_BEHAVIOR {
    INITIAL idle
    IN living_room {
        idle -> record: motion_detection
        record -> idle: deactivate
    }
    IN bedroom {
        idle -> alarm_on: motion_detection
        alarm_on -> idle: deactivate
    }
}
COMPONENT_TO_STATE_MAP {
    SENSORS: alarm_on record
    CAMERAS: record
}"""
```

What hardware is installed and in what room?

What happens when a motion is detected in the living room?

…and in the bedroom?

What hardware is used in the different states of the system?

WHAT MAKES A DSL SYNTAX GOOD?

1. **Hierarchical Structure**
   - The DSL uses nested blocks (`SYSTEM`, `SYSTEM_BEHAVIOR`, `COMPONENT_TO_STATE_MAP`) to organise related elements.
   - This reflects a real-world system configuration.
2. **Clarity and Readability**
   - Keywords like `SYSTEM`, `SYSTEM_BEHAVIOR`, and `COMPONENT_TO_STATE_MAP` make the DSL self-explanatory and all in capital letters. Although improvements are possible.
3. **Concise Commands**
   - State transitions are expressed succinctly, such as `source_state -> target_state: event` (e.g.,`idle -> alarm_on: motion_detection`).

WRITE A GRAMMAR FOR THE FOLLOWING DSL

```
program = """
SYSTEM {
    CAMERAS { living_room  }
    SENSORS { living_room bedroom }
}
SYSTEM_BEHAVIOR {
    INITIAL idle
    IN living_room {
        idle -> record: motion_detection
        record -> idle: deactivate
    }
    IN bedroom {
        idle -> alarm_on: motion_detection
        alarm_on -> idle: deactivate
    }
}
COMPONENT_TO_STATE_MAP {
    SENSORS: alarm_on record
    CAMERAS: record
}"""
```

|	choice
?	optional
*	zero or more times
+	one or more times
(…)	grouping

Duration: take 10 minutes to discuss with your partner and then share with the others

Think → Pair → Share

```
grammar = """
start: system (rules)? (maps)?

system: "SYSTEM" "{" (camera sensor | sensor camera) "}"
camera: "CAMERAS" "{" ID+ "}“
sensor: "SENSORS" "{" ID+ "}“

rules: "SYSTEM_BEHAVIOR" "{" rule+ "}"
rule: "INITIAL" ID -> initial | "IN" ID "{" action+ "}"
action: transition | transition ":" event
transition: ID "->" ID
event: "after" NUMBER "second"
       | ID

maps: "COMPONENT_TO_STATE_MAP" "{" map+ "}"
map: "SENSORS" ":" ID+
   | "CAMERAS" ":" ID+

%import common.CNAME -> ID
%import common.NUMBER
%import common.WS
%ignore WS
"""
```

```
program = """
SYSTEM {
    CAMERAS { living_room  }
    SENSORS { living_room bedroom }
}
SYSTEM_BEHAVIOR {
    INITIAL idle
    IN living_room {
        idle -> record: motion_detection
        record -> idle: deactivate
    }
    IN bedroom {
        idle -> alarm_on: motion_detection
        alarm_on -> idle: deactivate
    }
}
COMPONENT_TO_STATE_MAP {
    SENSORS: alarm_on record
    CAMERAS: record
}"""
```

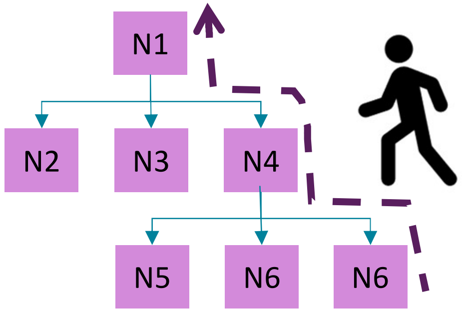
WHAT MAKES A GRAMMAR GOOD?

Clarity	Makes the DSL easier to understand and use.
Unambiguity	Ensures reliable parsing and interpretation.
Modularity	Promotes maintainability and reuse of grammar components.
Extensibility	Allows new features to be added without breaking existing rules.
Compactness	Avoids redundancy and keeps parsing efficient.
Domain Constraints	Validates domain-specific requirements directly in the grammar.
Consistency	Enhances readability and maintainability.

VALIDATION AND GENERATION

Powered by industry and academia

THE LARK TRANSFORMER (USES THE VISITOR DESIGN PATTERN)



The Visitor Pattern lets you separate algorithms from the objects they operate on.

Think of it as an inspector visiting different parts of your code structure depth-first.

Class **Transformer:** **start() : pass**
Class **MyTransformer (Transformer)**: start(): new_process()

THE LARK TRANSFORMER (REAL EXAMPLE)

Inherit from the base Transformer

```
# Pretty-printer transformer
class PrettyPrinter(Transformer):
    def start(self, items):
        system, rules, maps = items
        return f"{system}\n{rules}\n{maps}"

    def system(self, items):
        camera, sensor = items
        return f"SYSTEM:\n  {camera}\n  {sensor}"

                    ...

    def __default__(self, data, children, meta):
        return " ".join(children)

pretty_printer = PrettyPrinter()
result = pretty_printer.transform(tree)
print(result)
```

Has a method for each grammar rule

Each method signature takes self (the transformer class) and items (the children of the current transformed rule) as input parameter

__default__ method for all the rules for which the base Transformer is not overwritten

WHAT CAN WE DO WITH A TRANSFORMER?

- **Semantic Validation**: Checking the correctness of the DSL.
- **Error Handling**: Providing context-aware feedback for invalid DSL constructs.
- **Editor support**: Highlighting and formatting the code in an editor.
- **Output generation**: Converting the parse tree into meaningful outputs, such as configuration files, executable code, or other usable formats.

- We'll give two examples: **semantic validation** and **Output generation**

SEMANTIC VALIDATION

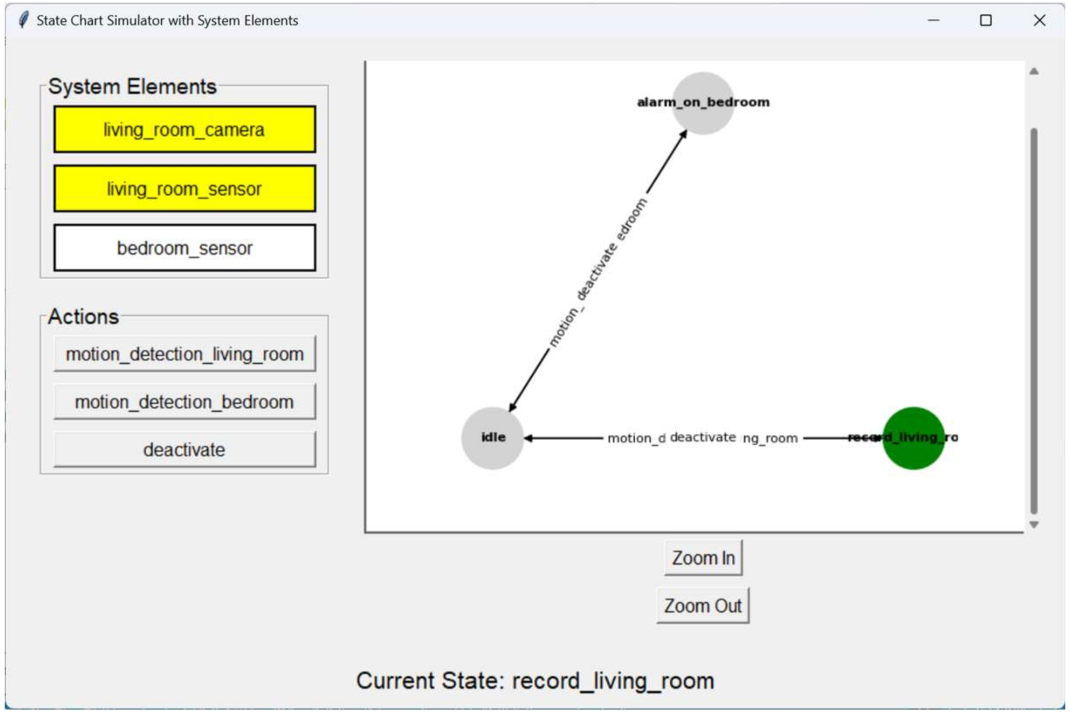
```
def check(program, grammar):

    class ExtractData(Transformer):
        def camera(self, rooms):
            return {"cameras": rooms}

        def sensor(self, rooms):
            return {"sensors": rooms}

          ...

    parser = Lark(grammar, parser='lalr’,
                  transformer=ExtractData())
    data = parser.parse(program)

    # Extracted data
    system = data[0]
    rules = data[1]
    maps = data[2]["maps"]
    cameras = system["cameras"]
    sensors = system["sensors"]
```

```
# Validation checks pseudo-code

room = all the room names

1. if room not in cameras or sensors:
2. errors(f"{room}’ not declared.")


1. if "record_video" in a room not in cameras:
2. errors(f"Action 'record_video' not
allowed.")


# Run the check
check(program, grammar)
```

Live demo

CODE GENERATION

```
class SystemRulesTransformer(Transformer):
    def __init__(self):
            ...
    def system(self, items):
        cameras, sensors = items
        self.sys.extend(cameras + sensors)
        return items
            ...
```




```
System: ['living_room_camera', 'living_room_sensor', 'bedroom_sensor’]
States: {'alarm_on_bedroom', 'record_living_room', 'idle’}
Transitions: {('idle', 'motion_detection_living_room'): 'record_living_room',
('record_living_room', 'deactivate'): 'idle', ('idle', 'motion_detection_bedroom'):
'alarm_on_bedroom', ('alarm_on_bedroom', 'deactivate'): 'idle’}
State to Elements: {'alarm_on_bedroom': ['bedroom_sensor'], 'record_living_room':
['living_room_sensor', 'living_room_camera']}
```

Live demo

CLOSING REMARKS

HAVE YOU REACHED THE OBJECTIVES?

- How many of you knew about Domain-specific languages before today?
- Do you understand the purpose and application areas of domain-specific languages?
- Do you feel capable of explaining the concepts and notations of formal grammar and parsing?
- Could you start designing basic DSLs to model software systems?

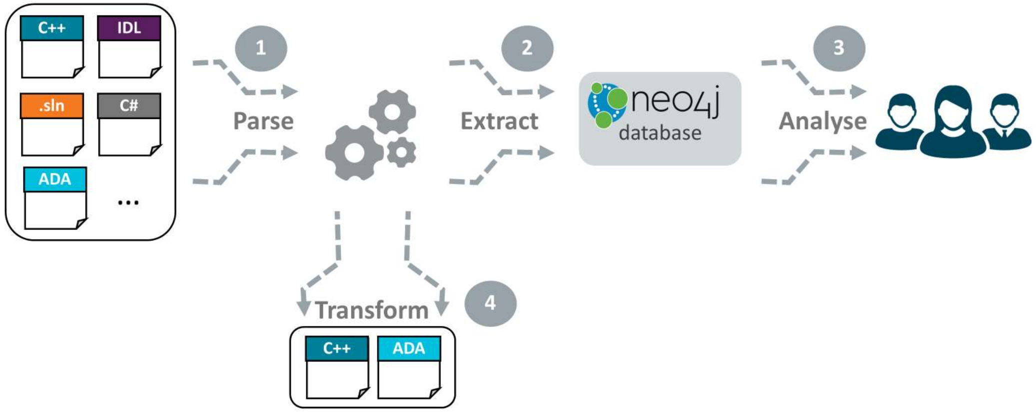
REFLECTION DOCUMENT

- Contents:
- Formulate your informed view on Model-Based Development for Software Systems
- Motivate this view based on your experiences in this course
    - (Optional) You may relate it to other (properly-referenced) experience/information sources
    - (Optional) You may relate it to your prior software development experiences
- Grading criteria:
    - Showing an understanding of model-based development for software systems
    - Providing an overarching view with supporting arguments (including your experiences in this course)
    - Referencing all used sources (facts, experiences, etc.) in an appropriate way

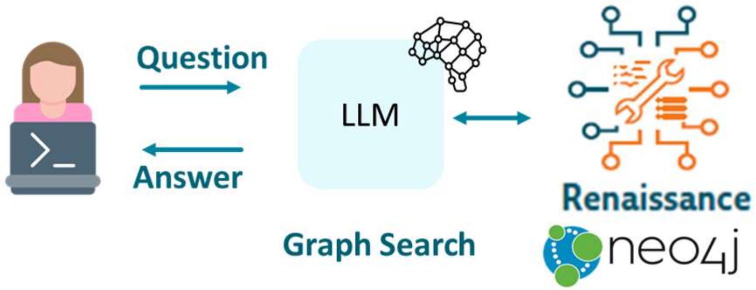
- Note: Individual assignment, to be submitted as PDF (Length: 1-page A4= 500 words)

MASTER INTERNSHIP POSSIBILITIES

RENAISSANCE: CODE ANALYSIS AND RESTRUCTURING



SMART CYPHER QUERY GENERATOR



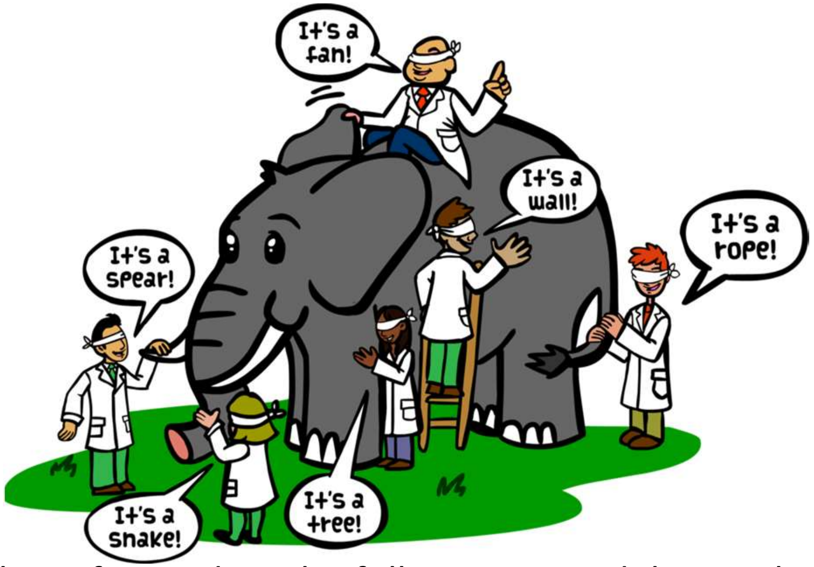
POSSIBLE TOPICS TO EXPLORE AROUND HYBRID SOLUTIONS COMBINING LLMS WITH DSLs:

1. Leveraging LLMs for Automated Code Analysis and Refactoring
2. Using LLMs to Ensure Requirements Compliance in Software Development
3. Using LLMs for program visualisation and understanding
4. Validating LLM-generated DSL code

If you are interested, contact me at:

rosilde.corvino@tno.nl

MODELING FOR A SPECIFIC PURPOSE



- In this course, we have focused on the following 3 modeling techniques:
  - Unified Modeling Language (UML) → Use cases / Structure of System / Sequence of event
  - Finite-State Machines (FSM) → Behavior of System / Components / Interfaces
  - Domain-Specific Languages (DSL) → Specific aspects of the structure or the behaviours

SEE YOU AT THE LAB ☺