BTech Project 2020

Group No. 7

A HIGH PRECISION CODE SUMMARIZER BASED ON UNIVERSAL ABSTRACT SYNTAX TREES

Guide:

1. Dr. Vahida Z. Attar,
   HOD of Comp and IT Dept,
   COEP
2. Mr. Shantanu Bedarkar,
   Embold Technologies
3. Mr. Pranay Jain,
   Embold Technologies

Group members:

1. Shreya Mehta – 111603035
2. Sneha Patil – 111603047
3. Nikita Shirguppi – 111603057

Contents:

INTRODUCTION:

WHAT IS CODE SUMMARIZER?

- An algorithm for summarizing the code in natural language.
- Extract important words from the code, understand the dependencies of the variables and their flow throughout the code, and then by deep learning techniques generate a text summary.

Sample Example:

Code Summary:

Prints Hello World

Sample example

```
public class ComputeGCD {
    public static void main(String[]
args) {
        int num1 = 55, num2 = 121;
        while (num1 != num2) {
            if(num1 > num2)
                num1 = num1 - num2;
            else
                num2 = num2 - num1;
        }
        System.out.printf("GCD of given
numbers is: %d", num2);
    }
}
```

Code Summary:

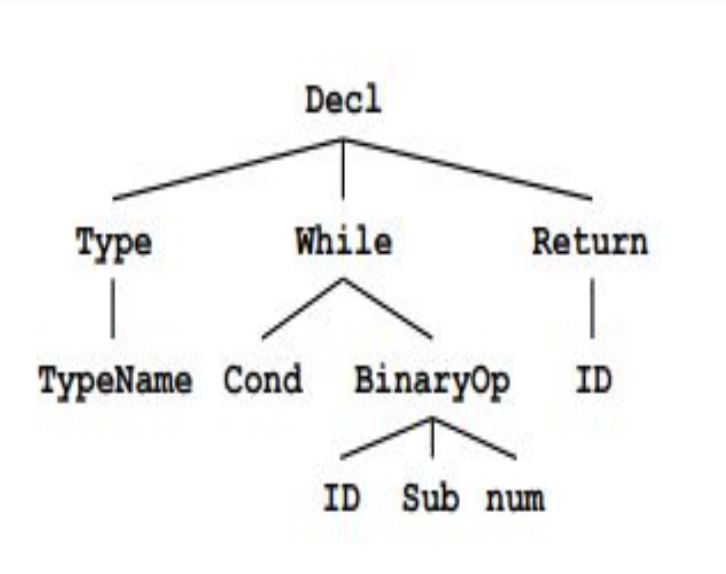
Prints GCD of given numbers

USE CASES

1. Useful for KT training of newly joined employees in a company.
2. Adding new feature to existing code.
3. For comprehending any God class.
4. For understanding legacy code.
5. For finding outlier functions/methods in a God class.
6. For providing overview/gist of any project.
7. Informational Retrieval on Web.
8. For open source developers.

Literature Survey:

1. Automatic Source Code Summarization with Extended Tree-LSTM:

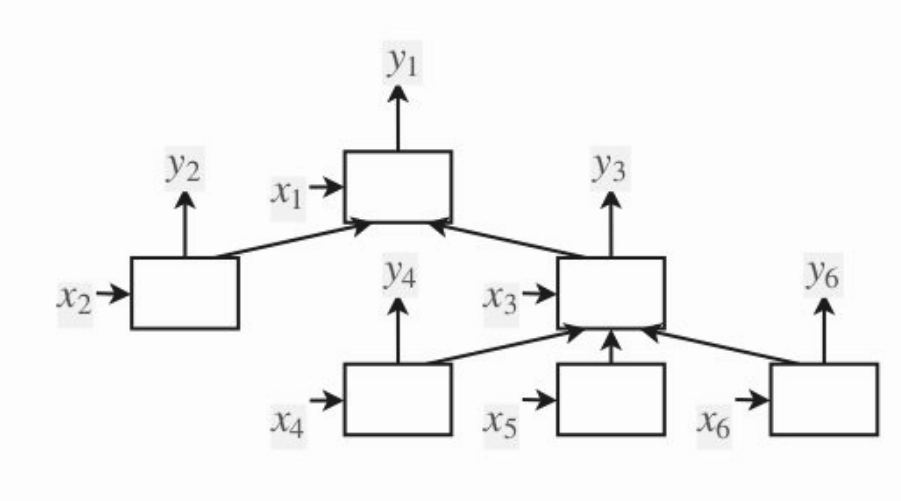


AST example

- AST(Abstract Syntax Tree) enables to use structural information of the source code.
- Tree LSTMs (Long Short Term Memory) are used to handle these structural data (as simple LSTMs are applicable only for the sequential type of data) but while applying these to ASTs, it cannot handle nodes which contain an arbitrary number of children, hence an extension of tree LSTM - Multiway Tree LSTMs are used here.

Gates in Tree LSTM:

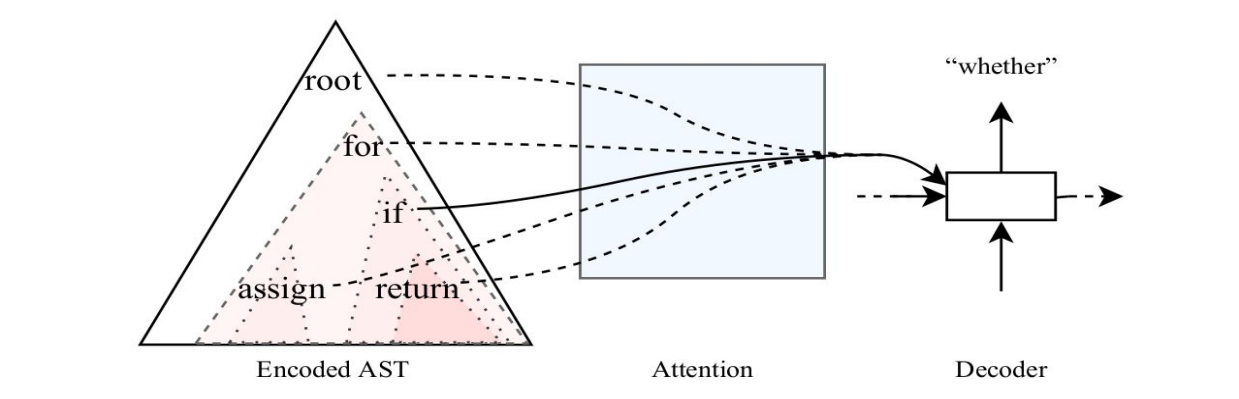
1. Forget gate
2. Input gate
3. Output gate



Tree LSTMS:

1. Child Sum Tree LSTM: interaction between children are not taken into consideration.
2. N-array LSTM: cannot handle an arbitrary number of children to the nodes.

Proposed Approach:



Encoder: encoder learns distributed representations of the nodes.

Attention mechanism: The attention mechanism focuses on subtrees of the AST.

Decoder: Decodes the hidden states in the encoder into a sentence in the target language.

Approach	BLEU-1	BLEU-2	BLEU-3	BLEU-4
CODE-NN	0.2779	0.2208	0.1974	0.1829
DeepCom	0.2894	0.2361	0.2132	0.1988
Multi-way (1-layered)	0.2968	**0.2431**	**0.2191**	**0.2040**
Multi-way (2-layered)	**0.2984**	0.2413	0.2170	0.2015
Child-sum	0.2968	0.2400	0.2153	0.1997
N-ary	0.2944	0.2366	0.2117	0.1959
Transformer	0.1173	0.0583	0.0378	0.0249

Conclusion:

Summaries generated by using Multiway Tree LSTMs (with one-way encoder) is more expressive than other methods like N-array Tree LSTM, Child-sum Tree LSTM, Multiway Tree LSTMS (2 layer), CODE-NN, DeepCOM, etc.

2. Summarizing Source Code using a Neural Attention Model

1. CODE-NN uses Long Short Term Memory (LSTM) networks with attention to produce sentences that describe C# code snippets and SQL queries. CODE-NN is trained on a new corpus that is automatically collected from StackOverflow.
2. CODE-NN generates a NL summary of source code snippets (GEN task).
3. CODE-NN also performs the inverse task to retrieve source code given a question in NL (RET task).

Defining the 2 tasks :

Let UC be the set of all code snippets and UN be the set of all summaries in NL.

For a training corpus with J code snippet and summary pairs :

$(c_j , n_j), 1 \leq j \leq J, c_j \in UC, n_j \in UN ,$

GEN For a given code snippet $c \in UC$: the goal is to produce a NL sentence $n* \in UN$ that maximizes some scoring function

$s \in (UC \times UN \rightarrow R)$: $n* = \text{argmax}\ n\ s(c, n)$

RET We also use the scoring function s to retrieve the highest scoring code snippet $c * j$ from our training corpus, given a NL question $n \in UN$: $c * j = \text{argmax}\ c_j\ s(c_j , n), 1 \leq j \leq J.$

In this work, s is computed using an LSTM neural attention model

Building the dataset :

- collected data from StackOverflow (SO), a popular website for posting programming-related questions.
- CLEANING : trained a semi-supervised classifier to filter titles and gather (title, query) pairs and SQL pairs. 80 % used for training, 10%for validation and 10% for testing.
- PARSING : ANTLR parser for C# and python sqlparse for SQL.
- DATA STATISTICS : 1.C# snippets – 38 tokens long
  2. Queries – 46 tokens long,
  3. Titles are 9 –12 words long.
- HUMAN ANNOTATIONS : To address limitation of available tokens, humans were made to give short summaries of code snippets and this data set was taken into consideration during building the final dataset.

Accuracy :

ERROR	% CASES
Correct	37
Redundancy	27
Missing Info	26
Out of context	20

Loopholes:

- Only the lexical approach was taken into consideration.
- High Accuracy could not be achieved .

3. TASSAL: Autofolding for Source Code Summarization

- Automatically creates a summary of each source file in a project by folding its least salient code regions.
- While modern code editors do provide code folding to selectively hide blocks of code, it is impractical to use as folding decisions must be made manually or based on simple rules.
- (TASSAL) Tree-based Autofolding Software Summarization ALgorithm, is based on optimizing the similarity between a summary of a source file and the source file itself. This is the first content-based autofolding method for code summarization.

bigbluebutton/bigbluebu

Topic Model Vector Space Model

Compression ratio:

50

- SipDateHeader.java
- SipHeaders.java
- StatusLine.java
- SubjectHeader.java
- SubscriptionStateHeader.java
- SupportedHeader.java
- ToHeader.java
- UnsupportedHeader.java
- UserAgentHeader.java
- ViaHeader.java
- WwwAuthenticateHeader.java
- message
  - BaseMessage.java
  - BaseMessageFactory.java
  - BaseMessageOtp.java
  - BaseSipMethods.java
  - BaseSipResponses.java

```
1  /*...*/
23
24 package org.zoolu.sip.header;
25
26
27
28
29 /** SIP Status-line, i.e. the first line of a response message */
30 public class StatusLine
31 {
32    protected int code;
33    protected String reason;
34
35    /** Construct StatusLine  */
36    public StatusLine(int c, String r)
37    {  code=c;
38       reason=r;
39    }
40
41    /** Create a new copy of the request-line*/
42    public Object clone()
43    {...}
45
46    /** Indicates whether some other Object is "equal to" this StatusLine */
47    public boolean equals(Object obj)
48    {...}
56
57    public String toString()
58    {...}
60
61    public int getCode()
62    {...}
64
65    public String getReason()
66    {...}
68 }
69
```

OUTLINE of TASSAL :

- Takes as input a set of source files along with a desired compression ratio.
- outputs a summary of each file where uninformative regions of code have been folded
- STEPS :

1. Parses the code's AST to obtain suitable regions to fold.
2. Applies a source code language model to each foldable region.
3. Identify for every source file, which tokens specifically characterize the file, as opposed to project-specific or Java-generic tokens that are not as informative for understanding the file.
4. Determines the most uninformative regions to fold while achieving the desired level of compression.

BaseLines of TASSAL :

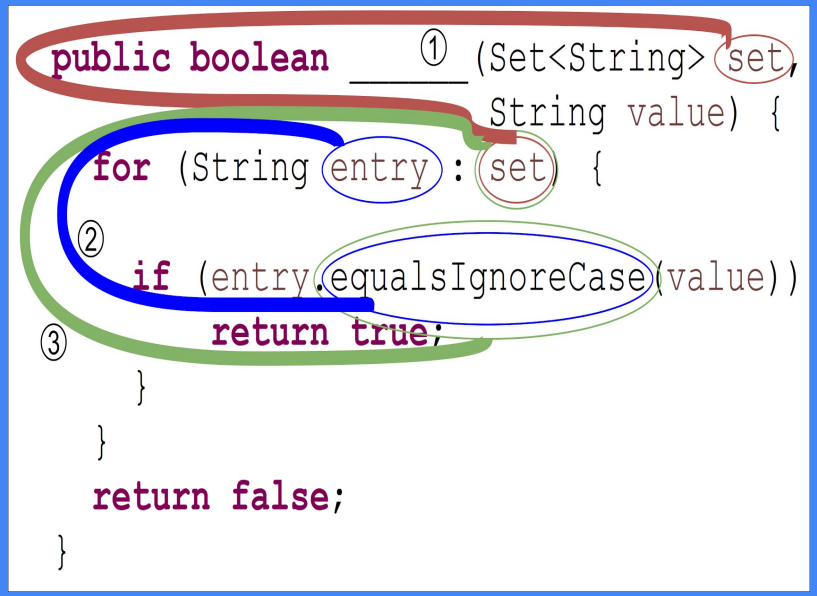
1. Shallowest unfold – the shallowest available foldable region in the AST first, choosing randomly if there is more than one.
2. Largest unfold – the largest available foldable region first, as measured by the number of tokens, breaking ties randomly.
3. Javadoc – first unfold all Javadoc comments (in random order) and then fallback at random to an available foldable region, unfolding method blocks last.

	Conciseness		Usefulness	
Summary	Mean	St. dev.	Mean	St. dev.
TASSAL	**3.27**	**1.01**	**3.18**	**0.97**
Javadocs*	3.07	1.03	2.69	1.09
Shallowest*	2.97	1.05	2.50	1.15
Largest*	3.08	1.07	2.67	1.06

4. CODE 2 SEQ

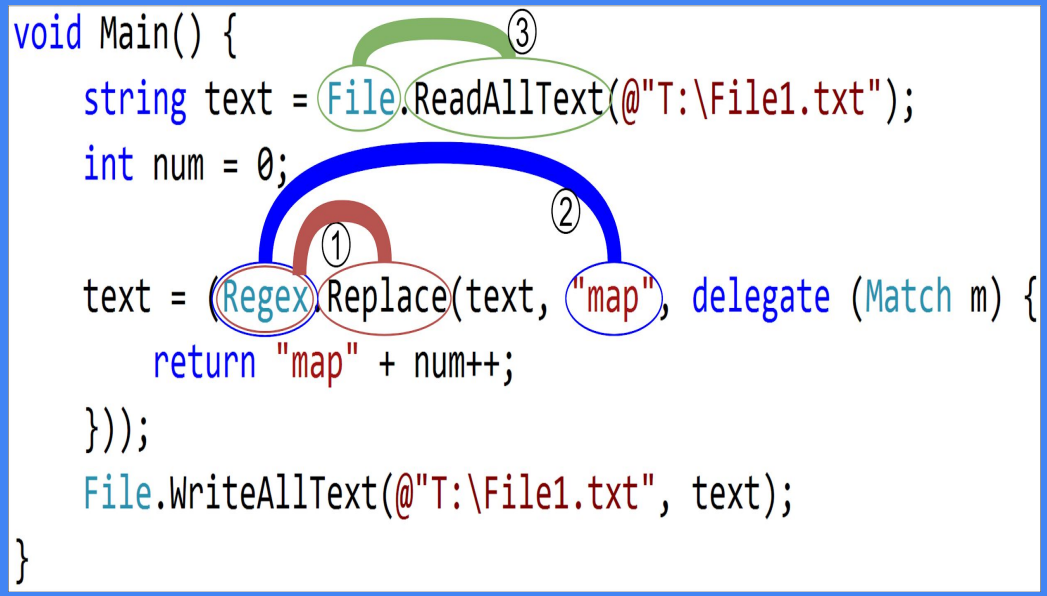
- The model leverages the syntactic structure of programming languages to better encode source code.
- Model represents the code as a set of compositional paths in its Abstract Syntax Tree(AST).
- Use Attention to select relevant paths while decoding.

Code Summarization in JAVA



contains① ignore② case③

Code Captioning in C#



replace① a string② in a text③ file③

The highlighted paths in each example are the top-attended paths in each decoding step.

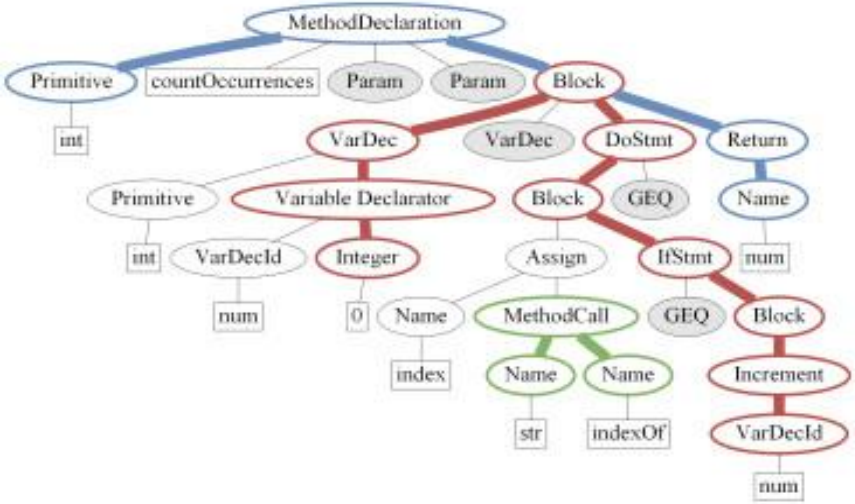
Representing Code As AST

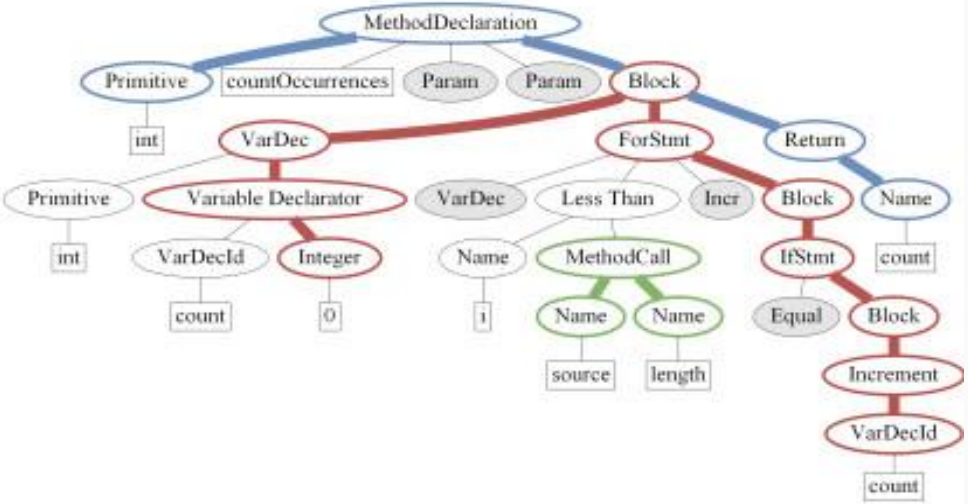
- An AST uniquely represents a source code snippet in a given language and grammar.
- All pairwise paths for a given AST between terminals are considered and represented as sequence of terminal and non-terminal nodes.
- We then use these paths with their terminals' values to represent the code snippet itself.
- Consider example on the next slide two Java Methods which count occurence of a string.

#Note: They have exactly same functionality, but a different implementation, and therefore different surface forms.

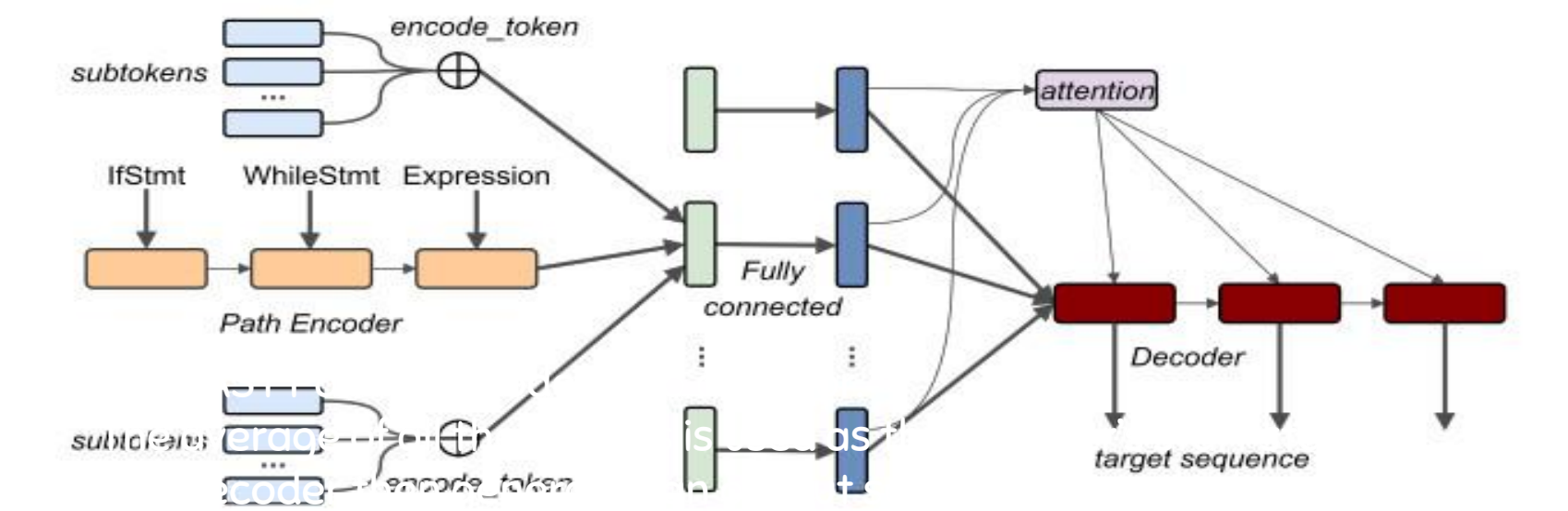
```java
int countOccurrences(String str, char ch) {
    int num = 0;
    int index = -1;
    do {
        index = str.indexOf(ch, index + 1);
        if (index >= 0) {
            num++;
        }
    } while (index >= 0);
    return num;
}
```

```java
int countOccurrences(String source, char value) {
    int count = 0;
    for (int i = 0; i < source.length(); i++) {
        if (source.charAt(i) == value) {
            count++;
        }
    }
    return count;
}
```





Model Architecture



Variations on the code2seq model to understand the importance of its different components

Model	Precision	Recall	F1	ΔF1
code2seq (original model)	**60.67**	**47.41**	**53.23**	
No AST nodes (only tokens)	55.51	43.11	48.53	-4.70
No decoder	47.99	28.96	36.12	-17.11
No token splitting	48.53	34.80	40.53	-12.70
No tokens (only AST nodes)	33.78	21.23	26.07	-27.16
No attention	57.00	41.89	48.29	-4.94
No random (sample k paths in advance)	59.08	44.07	50.49	-2.74

1. No AST nodes – instead of encoding an AST path using an LSTM, take only the first and last terminal values to construct an input vector
2. No decoder – no sequential decoding; instead, predict the target sequence as a single sym–
bol using a single softmax layer.
3. No token splitting – no sub–token encoding; instead, embed the full token.

Variations on the code2seq model to understand the importance of the different components

Model	Precision	Recall	F1	ΔF1
code2seq (original model)	**60.67**	**47.41**	**53.23**	
No AST nodes (only tokens)	55.51	43.11	48.53	-4.70
No decoder	47.99	28.96	36.12	-17.11
No token splitting	48.53	34.80	40.53	-12.70
No tokens (only AST nodes)	33.78	21.23	26.07	-27.16
No attention	57.00	41.89	48.29	-4.94
No random (sample k paths in advance)	59.08	44.07	50.49	-2.74

4. No tokens – use only the AST nodes without using the values associated with the terminals.
5. No attention – decode the target sequence given the initial decoder state, without attention.
6. No random – no re-sampling of k paths in each iteration; instead, sample in advance and use the same k paths for each example throughout the training process.

5.Automatic Source Code Summarization of Context for Java Methods

Behaviour of method

(How to use?)

Context of method ?

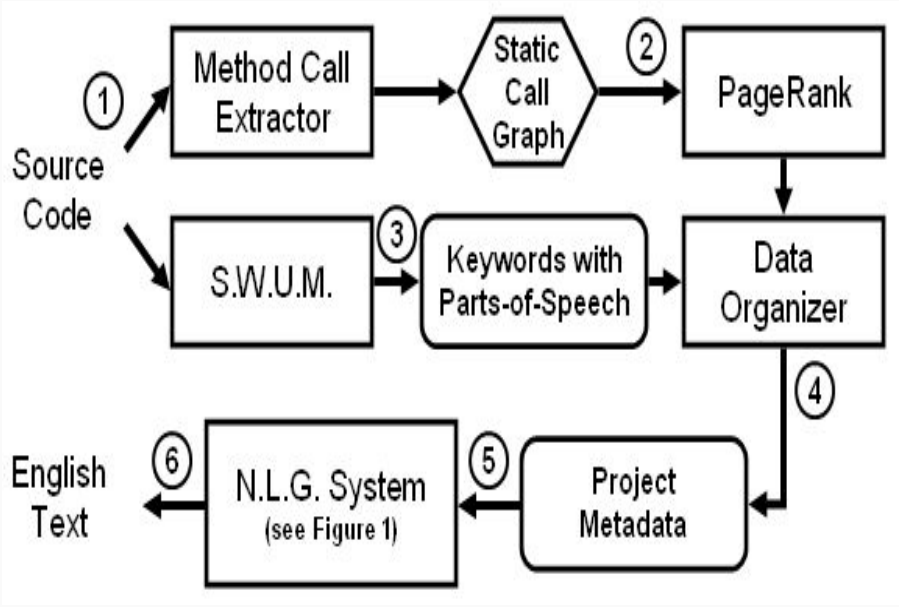
(Why does the method exist)

Example Method	**Summary**
1)StdXMLReader.read()	Reads a character.
Methods from context of example method	**Summary**
2)XMLUnit.skipWhitespace()	Skips whitespace from the reader.
3)XMLElement.addChild()	Adds a child element.
4)StdXMLBuilder.startElement()	This method is called when a new XML element is encountered.
5)StdXMLBuilder.addAttribute()	This method is called when a new attribute of an XML element is encountered.

"This method reads a character. That character is used in methods that add child XML elements and attributes of XML elements. Calls a method that skips whitespace."

Approach

1. Use PageRank to discover the most important methods in the given method's context.
2. Use data from SWUM to extract keywords about the actions performed by those most important methods.
3. Use a custom NLG system to generate English sentences describing for what the given method is used.



Research Gaps:

1. In the current research, either Lexical or Structural approaches are considered independently.
2. Git log is not considered.
3. Only method names are considered for predicting the summary.
4. Universal Abstract Trees are never considered.
5. Outliers like finding an independent isolated method in a God class.

Problem Statement:

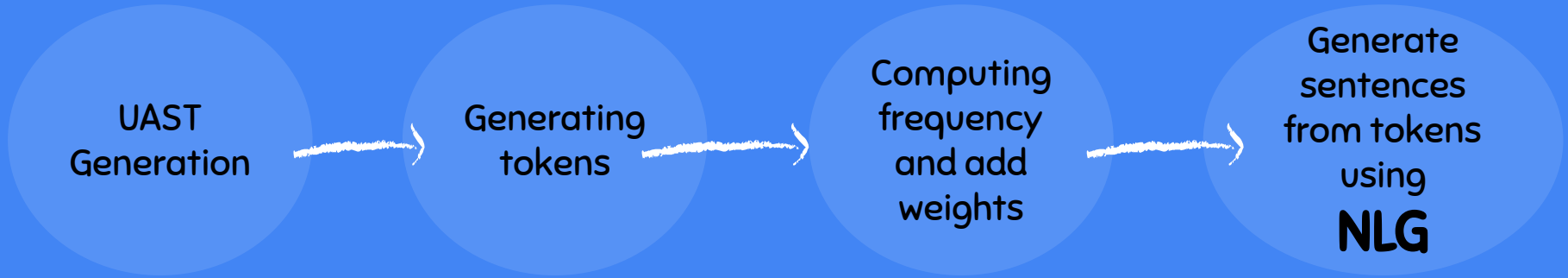
We propose an approach for a high precision code summarizer that uses the UAST (Universal Abstract Syntax Tree) to parse the code and select the best possible weighted path.

Objectives:

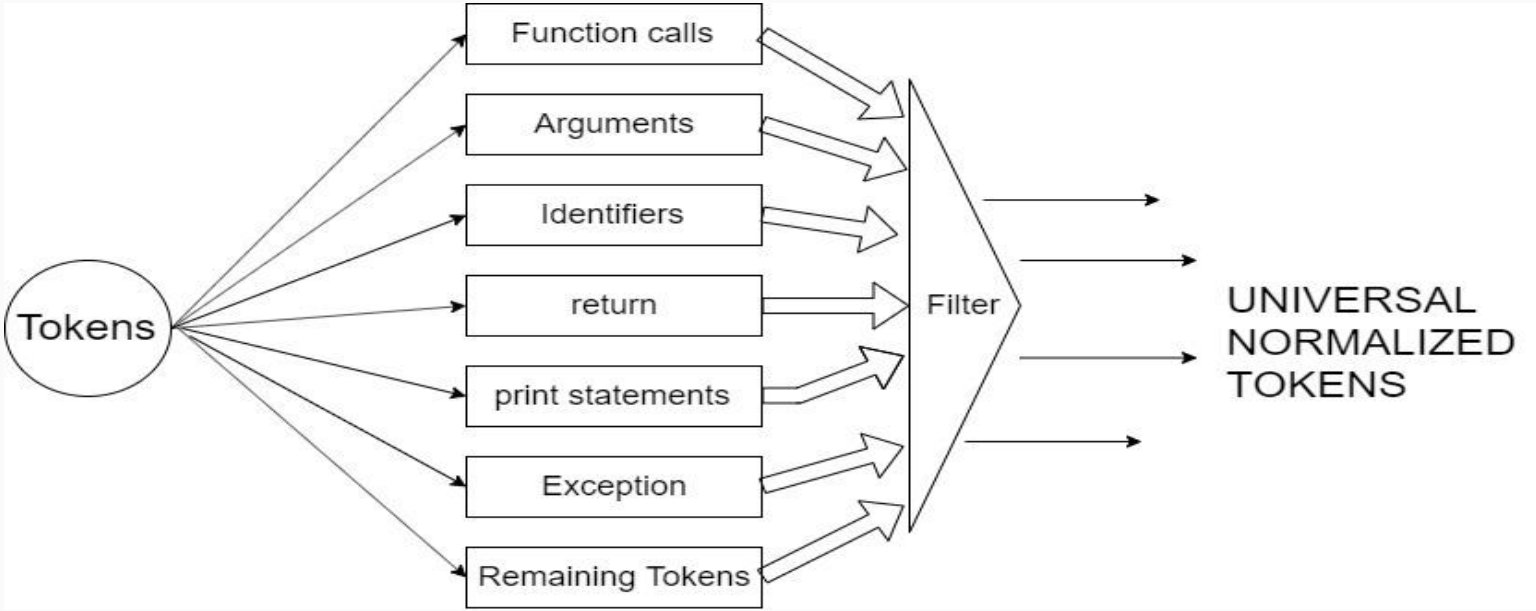
1. Implementing lexical approach for predicting summary.
2. Implementing structural approach for summarization.
3. Clubbing both the ways to increase the precision of the summary.
4. A completely orthogonal approach – using git log to provide summaries for the evolving code.

Methodology

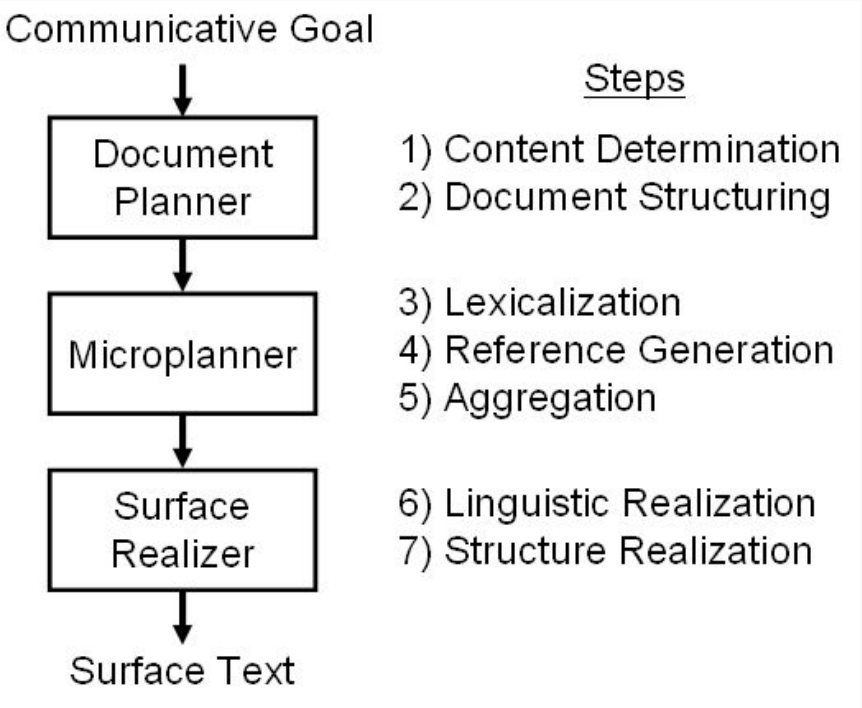
Lexical Approach



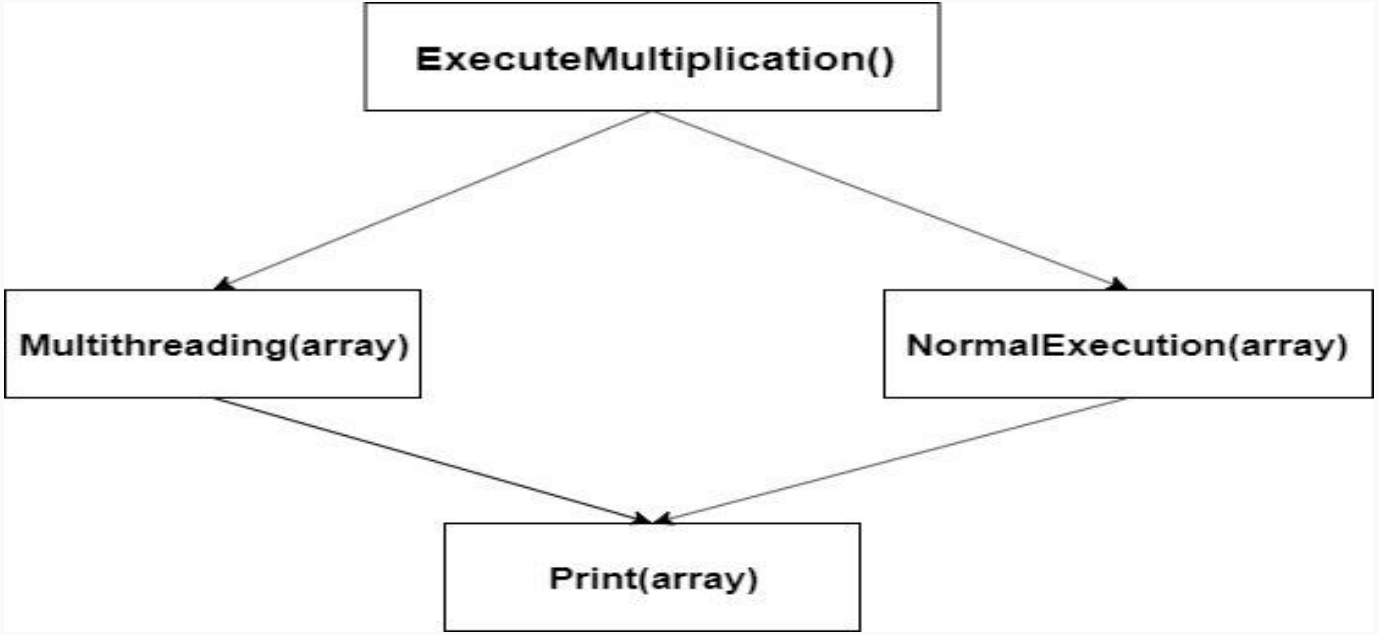
Token Generation And Normalization :



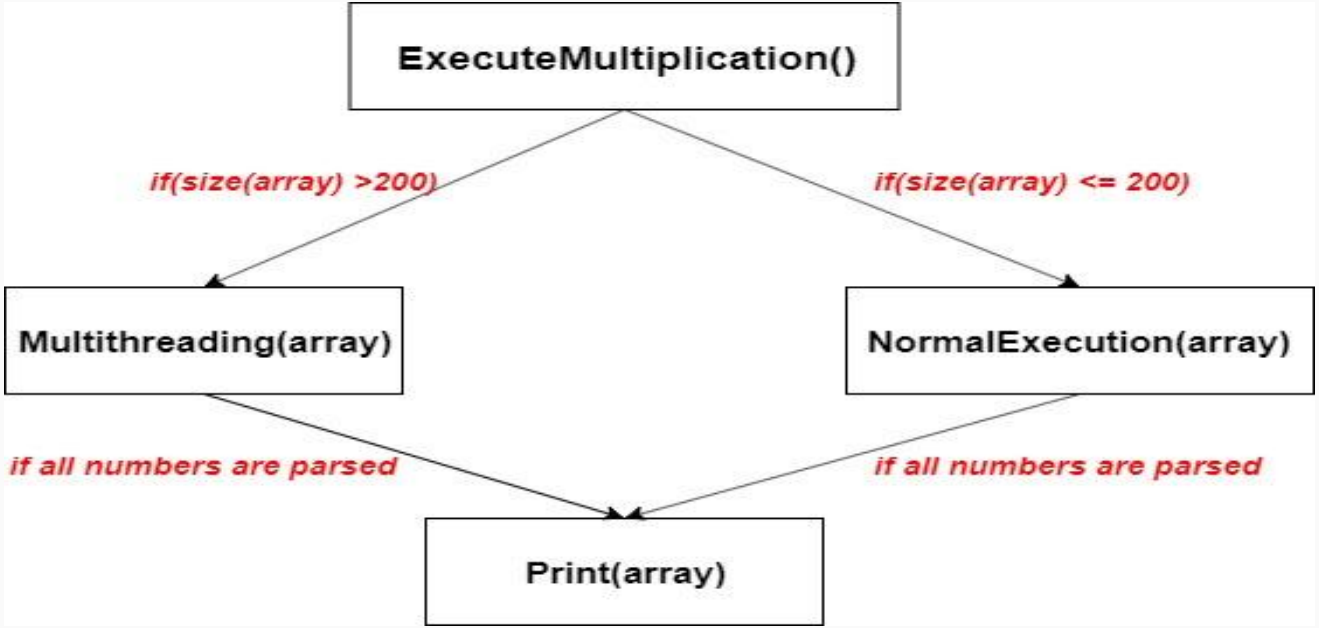
Natural Language Generation Systems



What lexical approach attempts to provide



Why do we need structural approach?



Why do we need structural approach?

```
public class ABC {
    public static void main(String[] args) {
        int num1 = 55, num2 = 121;
        while (num1 != num2) {
            if(num1 > num2)
                num1 = num1 - num2;
            else
                num2 = num2 - num1;
        }
        System.out.printf("%d", num2);
    }
}
```

Lexical approach fails here!

METHODOLOGY

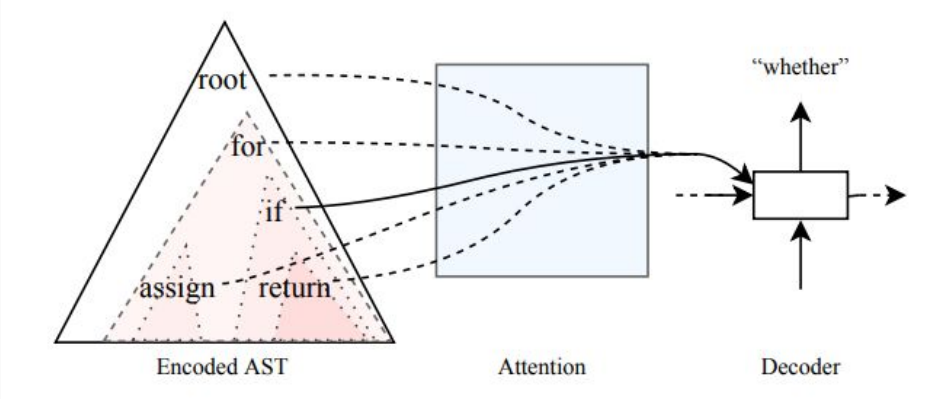
Structural approach:

UAST Generation → Generating operation specific tokens → Relating variables with their operations → Generate sentences from tokens

Methodology

Structural approach :

1. Use LONG SHORT TERM MEMORY (LSTM) – for long term dependencies
2. Will have 3 independent gates – input, forget, output.
3. Use Multiway Tree–LSTMs.
4. Use Encoder, Attention mechanism and Decoder.



Git Log approach:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

Example of Git LOG

Git Log approach:

Getting the tokens from the non evolved code

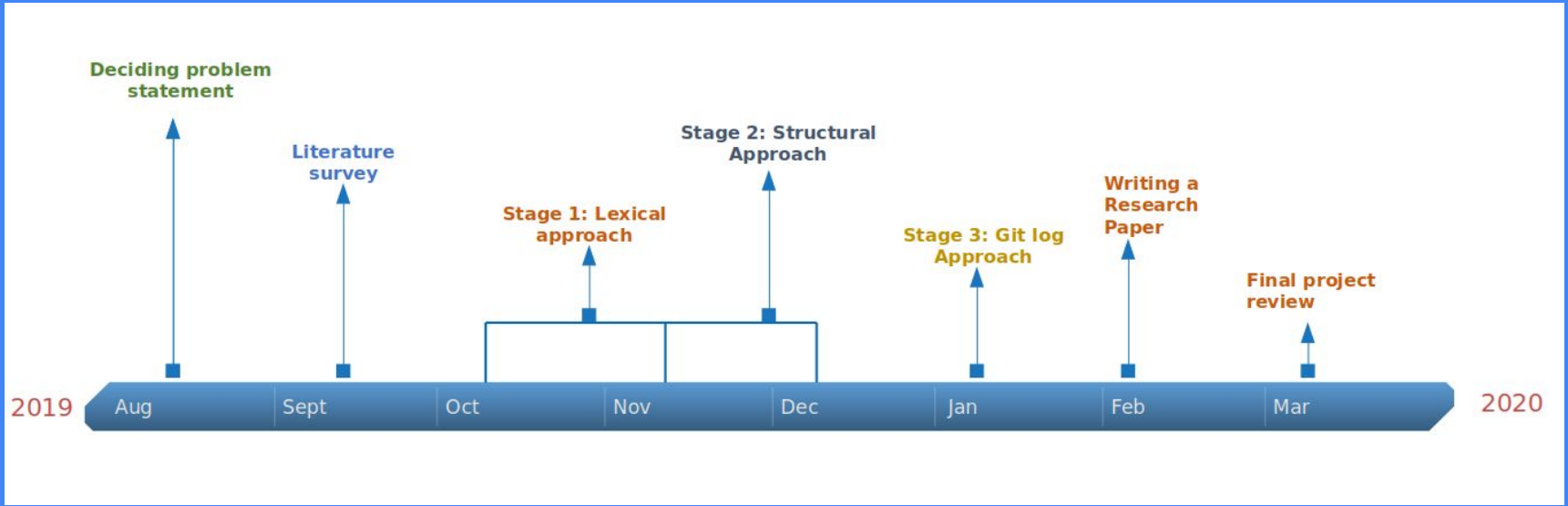
Getting the tokens from the evolved code

Using machine learning model, to generate tokens from commit message.

Difference between the tokens (evolved and non evolved)

Relation between this diff, and commit message

Timeline for overall project:



Thank You