AI3602 Data Mining Lecture 8

Learning From Code

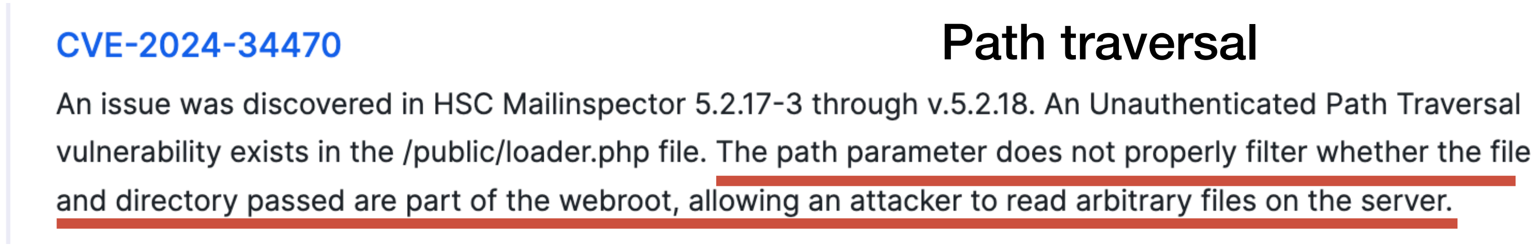
Liyao Xiang
xiangliyao.cn
Shanghai Jiao Tong University

- Why do we learn from source code?

Software is Buggy

- Software is everywhere: social media, websites, smartphone APP …
- Software has **bugs** and **security vulnerabilities**
  - E.g., uncontrollable acceleration in self-driving cars, personal information leakage from social media, glitches in smart thermostats lead to heat outage …

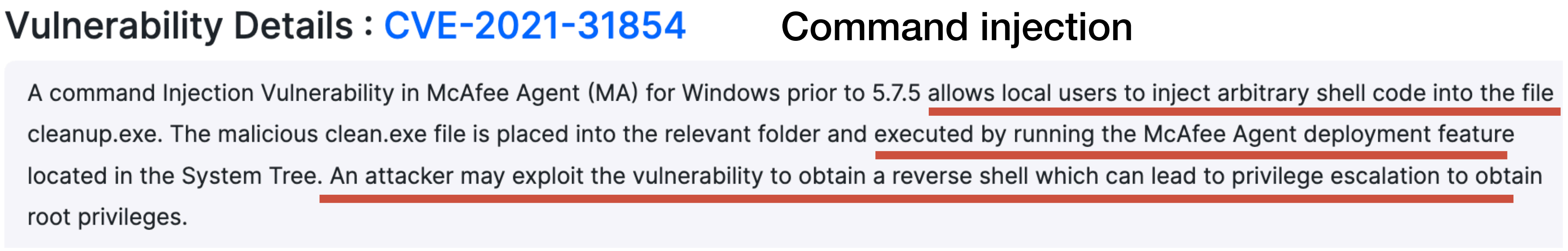
Path traversal



CVE-2024-34470

An issue was discovered in HSC Mailinspector 5.2.17-3 through v.5.2.18. An Unauthenticated Path Traversal vulnerability exists in the /public/loader.php file. The path parameter does not properly filter whether the file and directory passed are part of the webroot, allowing an attacker to read arbitrary files on the server.

Command injection



Vulnerability Details : CVE-2021-31854

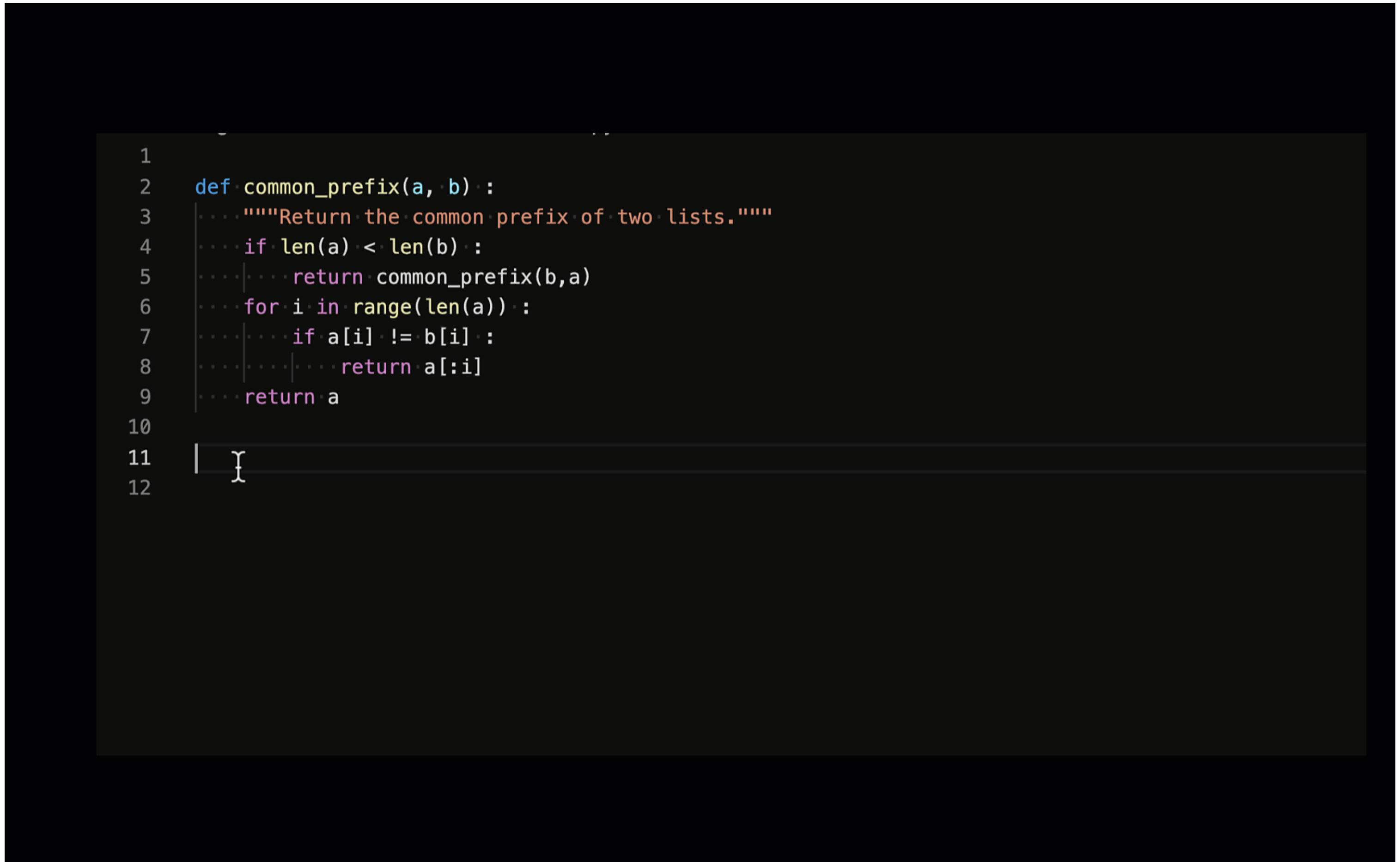
A command Injection Vulnerability in McAfee Agent (MA) for Windows prior to 5.7.5 allows local users to inject arbitrary shell code into the file cleanup.exe. The malicious clean.exe file is placed into the relevant folder and executed by running the McAfee Agent deployment feature located in the System Tree. An attacker may exploit the vulnerability to obtain a reverse shell which can lead to privilege escalation to obtain root privileges.

Program Analysis

- **Program analysis** is about analyzing software code to learn about its properties
  - Can summarize code functions
  - Can find bugs or security vulnerabilities
  - Can synthesize test cases
  - Can automatically produce patches for bugs …

Applications

- **Code generation**
  - Code prediction: method name prediction, next token prediction
  - Code snippet generation from description



```python
def common_prefix(a, b) :
    """Return the common prefix of two lists."""
    if len(a) < len(b) :
        return common_prefix(b,a)
    for i in range(len(a)) :
        if a[i] != b[i] :
            return a[:i]
    return a
```

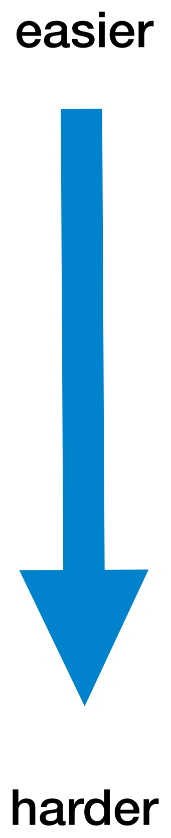
From Github Copilot

Applications

- **Code summarization**: use natural language to provide concise explanation of code's functioning
- **Code search**: the input is the natural language description or a code snippet, and the output is the best matching code snippets for input
  - Semantic alignment between input and code snippets
  - Outputs are retrieved from the code corpus

Applications

- **Clone detection**
    - Type 1: **identical** code fragments which may differ in white spaces, layouts, comments
    - Type 2: **identical** code fragments which may differ in variable names, constants, function names, identifiers, literals, types
    - Type 3: **syntactically similar** code fragments with added, deleted or modified statements
    - Type 4: **semantic similar** code fragments which may use different lexical and syntax to express equivalent semantic



Applications

- **Safety analysis**
  - Defect prediction: predict problematic code
  - Vulnerability prediction: prevent code from being attacked and improve the security of code
  - Malware classification: classify the program as a normal one or malware
- **Bug localization**: static bug location is determined by the control and data dependencies, dynamic bug location is determined by program execution
- **Program repair**: e.g., detect variable misuse and replace it with correct one

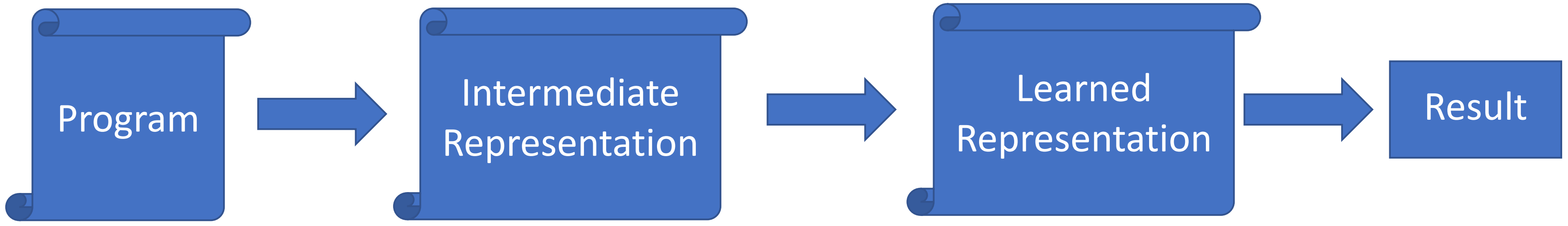
Automated Analysis of Programs

- Conventional tools include: mathematical formalisms, abstract interpretation, data flow analysis, pointer analysis, interprocedural analysis, symbolic execution …
- Today, we talk about **data mining** approach to analyze source code!
- First attempt: consider code as **plain text** sequences
  - ➡ but code’s structural information is missing, because code has two channels:
    - ❖ Formal (operational) channel — interpreters, compilers, etc. use the channel
    - ❖ Natural channel — used by humans for code comprehension and communication

Code Understanding Challenges

- Code structural modeling: how to **model structural information** in code effectively
  - ❖ **Generic representation** learning: learn **language-independent** code representations
  - ❖ **Task-specific adaptation**: choose and design specific **features** for downstream applications

Step 2:
- Obtain (latent) program representation
- Often re-uses ML infrastructure
- Usually produces a (set of) vectors



Step 1:
- Transform to ML-compatible representation
- Interface between code and ML
- Often re-uses **compiler** infrastructure

Step 3:
- Produce output useful to developer/program tool
- Interface between ML and code

- Step 1: Pre-processing

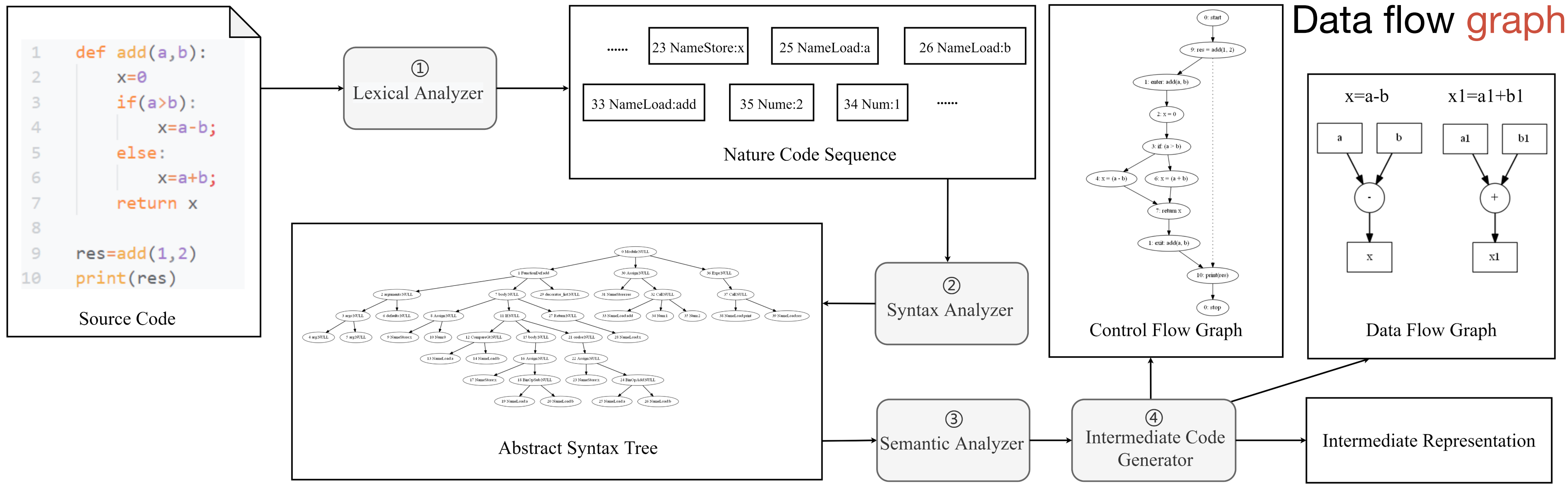
Code Pre-Processing

Lexical analyzer converts the code into a token-based sequence

Token sequence: type & value

Control flow graph

Data flow graph



```
def add(a,b):
    x=0
    if(a>b):
        x=a-b;
    else:
        x=a+b;
    return x

res=add(1,2)
print(res)
```

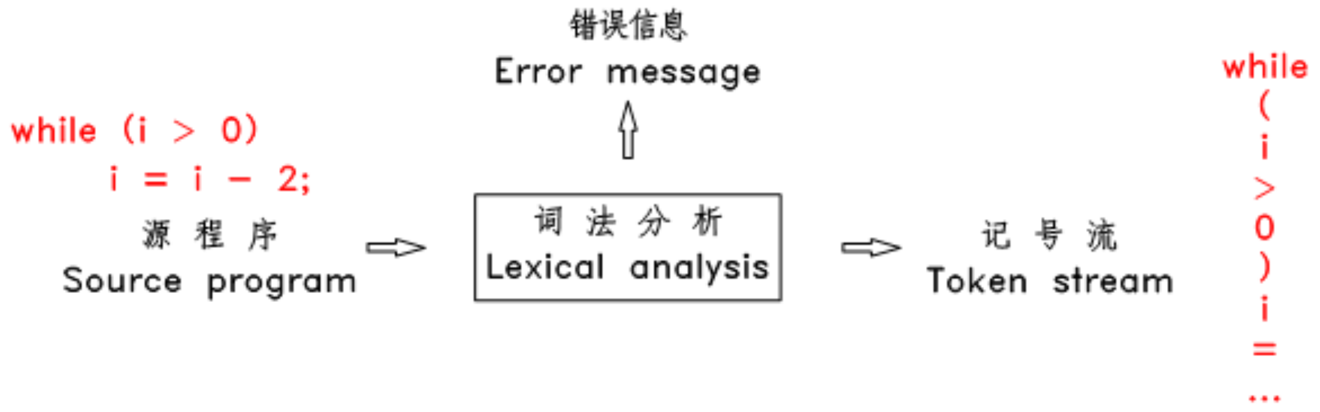
Syntax analyzer takes the tokens and produces an Abstract Syntax Tree

Semantic analyzer verifies semantic consistency

Intermediate **Representation Construction**

Lexical Analyzer

- Lexical analysis is the 1st phase of a compiler
  - Takes modified source code from language preprocessors in form of sentences
  - Breaks these syntaxes into a series of **tokens**, by removing any whitespace or comments in the source code



Lexical Analyzer

```
int value = 100;
```

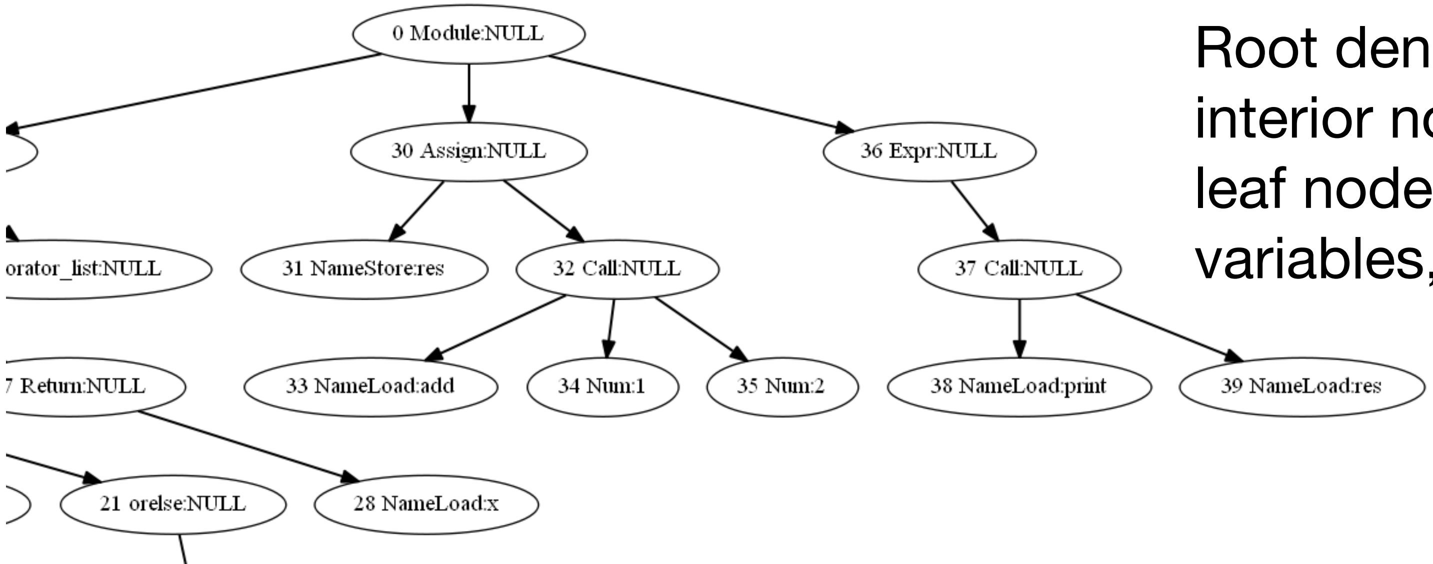
contains the tokens:

```
int        (keyword)
value (identifier)
=         (operator)
100       (constant)
 ;          (symbol)
```

- The lexical analyzer scans and identifies only a finite set of valid string/token/lexeme that belong to the language in hand
- It searches for the pattern defined by the language rules
- Output (type, value)

Syntax Analyzer

- Syntax Analysis or Parsing is the 2nd phase, i.e. after lexical analysis
  - checks whether the **given input is in the correct syntax** (of the language in which the input has been written)
  - Builds a **Parse tree or Syntax tree** which is constructed by using the pre-defined (context-free) grammar of the language and the input string



Root denotes the start symbol, interior nodes are non-terminals, leaf nodes are terminals (e.g., variables, identifiers …)

Semantic Analysis

- Semantic Analysis is the 3rd phase of compiler. It is **a collection of procedures** called by parser as required by grammar
  - makes sure that declarations and statements of program are semantically correct
  - uses syntax tree and symbol table to check whether the given program is semantically consistent with language definition

I. **Type Checking** – Ensures that data types are used in a way consistent with their definition

II. **Label Checking** – A program should contain labels references

III. **Flow Control Check** – Keeps a check that control structures are used in a proper manner (e.g.: no break statement outside a loop)

Semantic Analysis

```
float x = 10.1;
float y = x * 30;
```

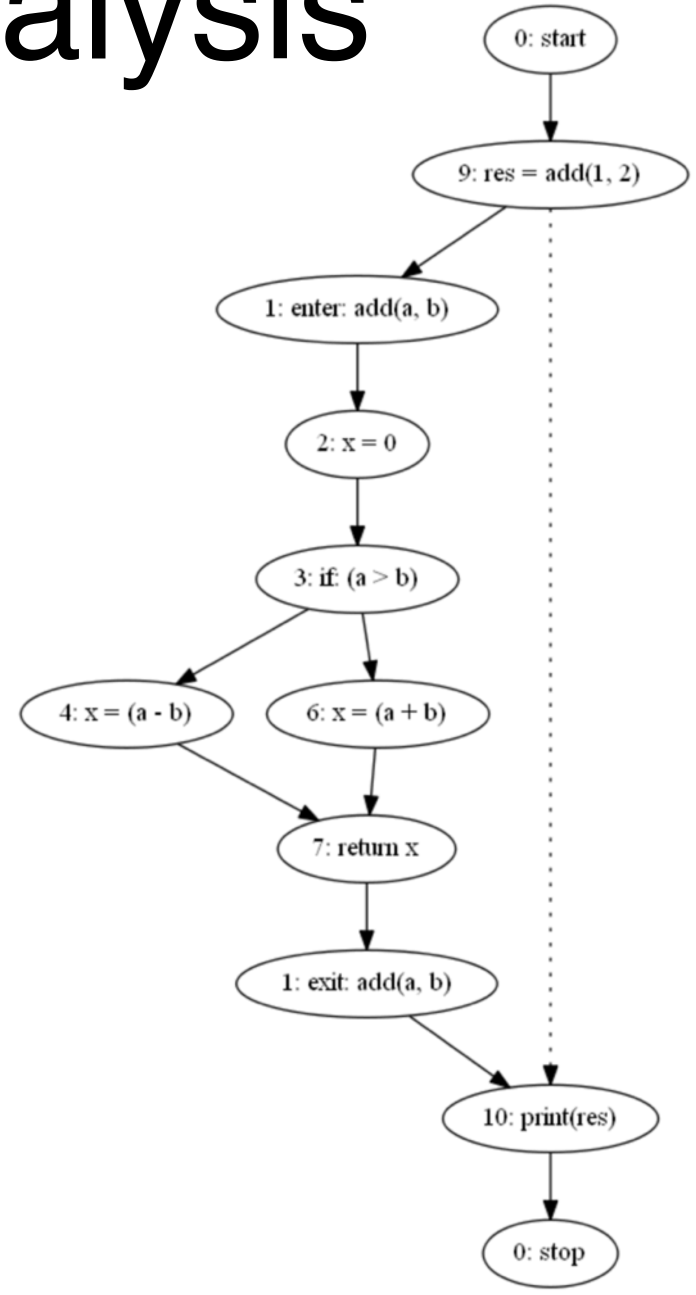
Integer 30 will be typecasted to float 30.0 before multiplication

```
int a = “value”;
```

Will not issue an error in lexical and syntax analysis phase, but will generate a semantic error as the type of assignment differs

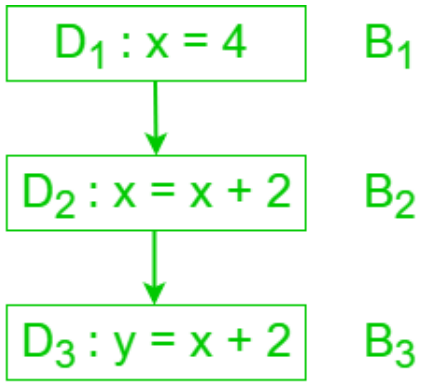
Control Flow Analysis

- A static-code-analysis technique for determining the control flow of a program
- Expressed as a control-flow graph
    - Nodes: basic code blocks
    - Edges: control flow paths



Data Flow Analysis

- Data flow tracks values of variables and expressions as they are computed and used throughout the program
  - Goal: identifying opportunities for **optimization** and **identifying** potential errors
  - Model program as a graph
    - Nodes: statements
    - Edges: data flow dependencies between statements



Data flow is propagated through the graph, using a set of rules and equations to compute the values of variables and expressions at each point in the program

D1 is a reachable for B2, but not for B3 as it is killed by D2

- Step 2: Learning

API Learning

- Obtain an API usage sequence based on an API-related natural language query
  - E.g., query "*parse XML files*" using JDK library, the desired API usage sequence is

    DocumentBuilderFactory.newInstance
    DocumentBuilderFactory.newDocumentBuilder
    DocumentBuilder.parse

  - In practice, usage patterns of API methods are not well documented, so that search engines are often inefficient and inaccurate for programming tasks
  - Statistical word alignment model is based on a *bag-of-words* assumption w/o considering the *sequence* of words and APIs

API Learning

- Formulate the problem as a **machine translation** problem: given a natural language query $x = x_1, \ldots, x_N$ where x_i is a keyword, we aim to translate it into an API sequence $y = y_1, \ldots, y_T$ where y_i is an API
- Requirement: Learn the sequence of words in a natural language query and the sequence of associated APIs
- Trains the language model that:
  - **encodes** each sequence of words (annotation) into a fixed-length context vector
  - **decodes** an API sequence based on the context vector

Language Model

- Regard **source code** as a special language and analyze it using statistical NLP techniques

- The language model is a **probabilistic model** that tells how likely a sentence would occur

$$Pr(y_1, ..., y_T) = \prod_{t=1}^{T} Pr(y_t | y_1, ..., y_{t-1})$$

A sequence of words

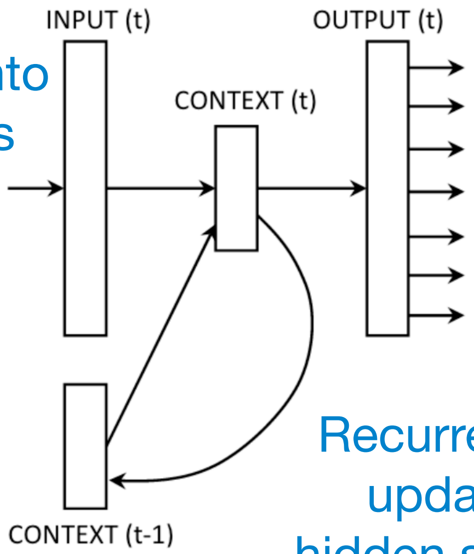
- N-gram model: the next word is conditioned only on the previous n-1 word

$$Pr(y_t | y_1, ..., y_{t-1}) \simeq Pr(y_t | y_{t-n+1}, ..., y_{t-1})$$

Neural Language Model

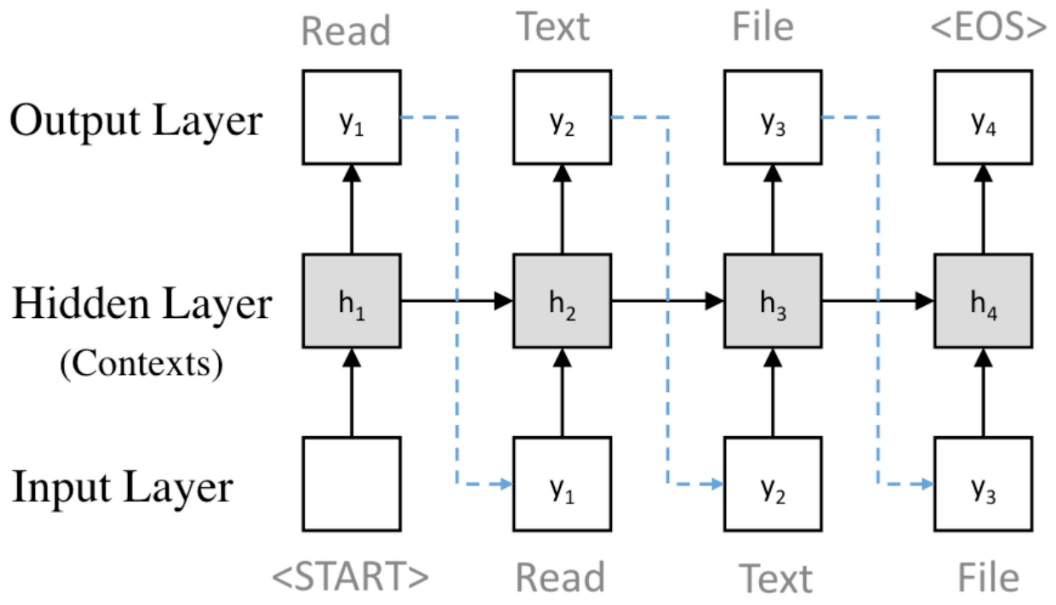
- Recurrent Neural Network (RNN)

Map words into vectors



Estimate probabilities for following word

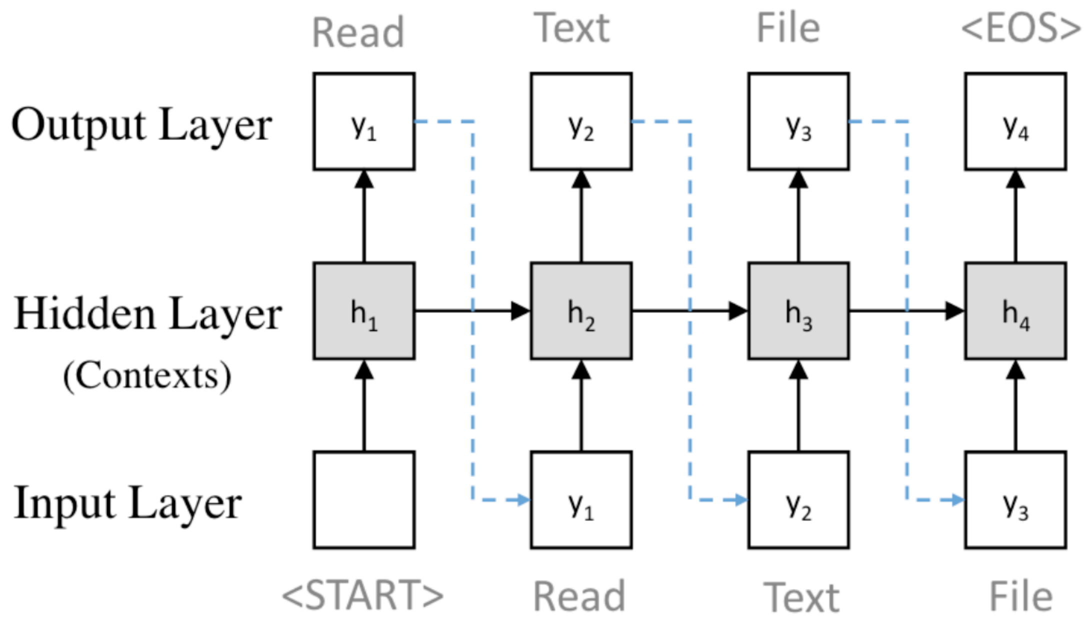
Recurrently update hidden states



Neural Language Model

- Recurrent Neural Network (RNN): three steps at each timestamp

Step 3: predict $Pr(y_{t+1}|y_1, ..., y_t) = g(\boldsymbol{h}_t)$



Step 2: generate the hidden state by previous hidden state and current input

$$\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{y}_t)$$

Step 1: map word to vector $\boldsymbol{y}_t = input(y_t)$

RNN Encoder-Decoder Model

- Generate a sentence of the target language given a sentence of the source language
  - Summarize the sequence of source words into a fixed-length context vector c

$$\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t)$$

$$\boldsymbol{c} = \boldsymbol{h}_{T_x}$$

f, p are RNNs

  - Generate the target sentence by sequentially predicting a word y_t conditioned on the source context c

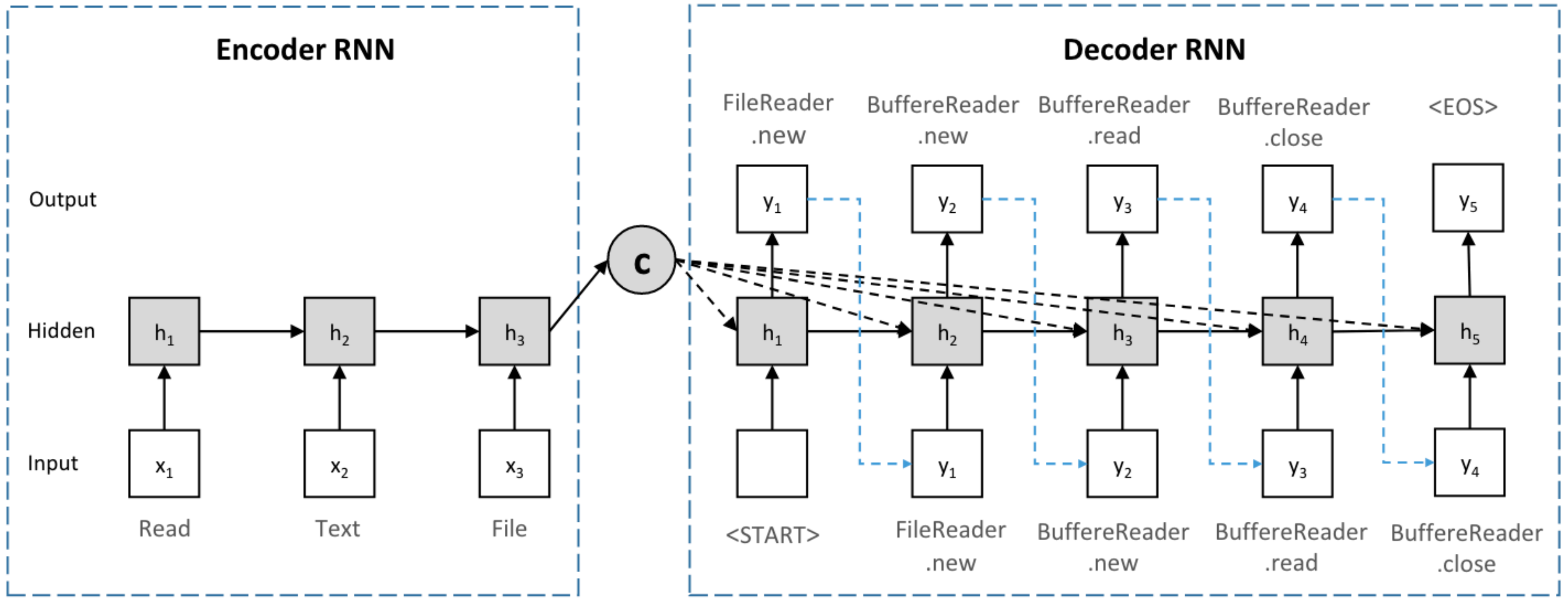
$$Pr(y) = \prod_{t=1}^{T} p(y_t | y_1, ..., y_{t-1}, \boldsymbol{c})$$

  - The RNN Encoder-Decoder model is trained to maximize the conditional log-likelihood

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \mathrm{cost}_{it} \qquad \mathrm{cost}_{it} = -\log p_\theta(y_{it} | x_i)$$

API Learning

- Translating sequence "*read text file*" to a sequence of APIs



Attention-Based Model

- Different parts of a query have *different importance* to an API in the target sequence
  - For query “*save file in default encoding*”, the word **file** is more important than **default** to the target API as the target API sequence is: *File.new, FileOutputStream.new, FileOutputStream.write, FileOutputStream.close*,
  - Attention-based RNN that selects the important parts from the input sequence for each target word

$$\boldsymbol{c}_j = \sum_{t=1}^{T_x} \alpha_{jt} \boldsymbol{h}_t$$

Weight between the hidden state h_t and the target word y_j

API Importance

- Different APIs have differed importance for a programming task
  - API *Logger.log* is widely used in many code snippets. But it cannot help understand the key procedures of a programming task
  - Consider the individual importance of APIs through IDF-based weighting: APIs that occur ubiquitously have lower weights

$$w_{idf}(y_t) = log(\frac{N}{n_{y_t}})$$

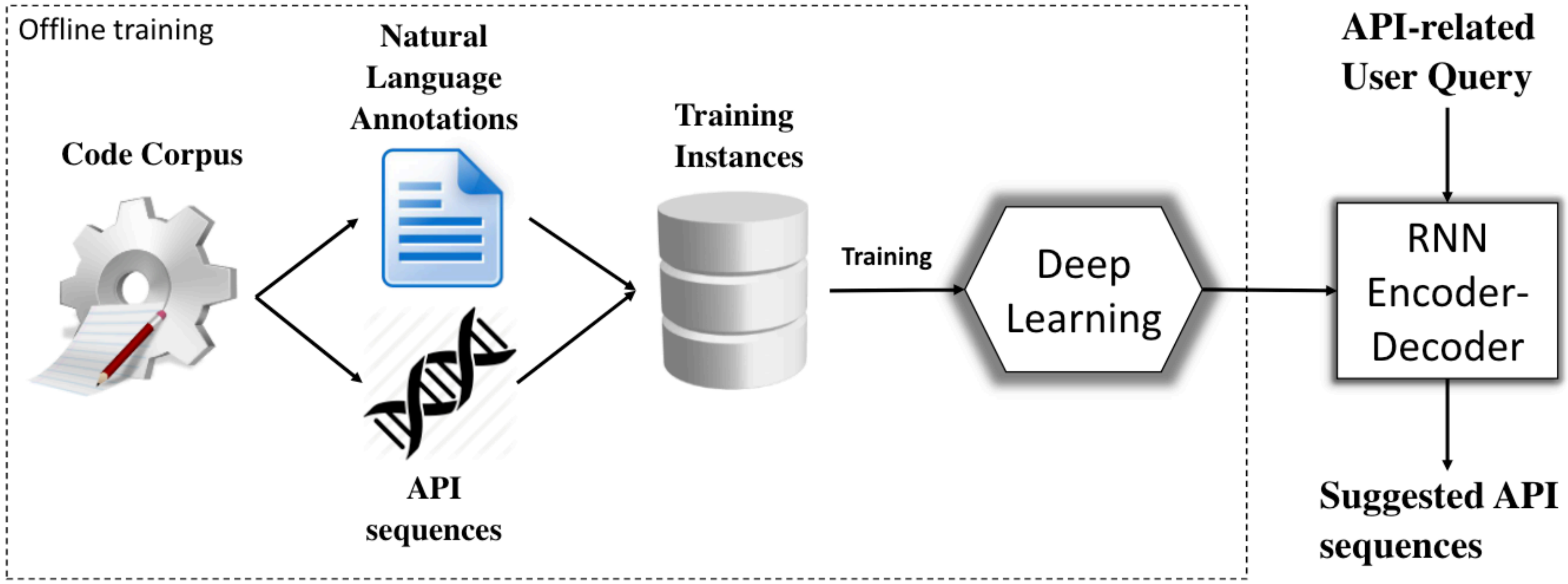
Total number of APIs

Occurences of API y_t

$$\mathrm{cost}_{it} = -\log p_\theta(y_{it}|x_i) - \lambda w_{idf}(y_t)$$

Deep API Learning

- Training on a large-scale corpus of annotated API sequences



Annotation Corpus

- Extract annotations: select the first sentence of the comment as the annotation
- Extract API usage sequences
  - ✦ Parse source code files into ASTs
  - ✦ Extract API sequence by traversing the AST of each method:
    - For each constructor invocation *new C( )*, append the API *C.new* to the API sequence
    - For each method call *o.m( )* where o is an instance of a JDK class C, append the API *C.m* to the API sequence
    - For a method call passed as a parameter, append the method before the calling method, e.g., for *o1.m1(o2.m2( ),o3.m3( ))*, we produce a sequence *C2.m2-C3.m3-C1.m1*

Annotation Corpus

✦ Extract API sequence by traversing the AST of each method:

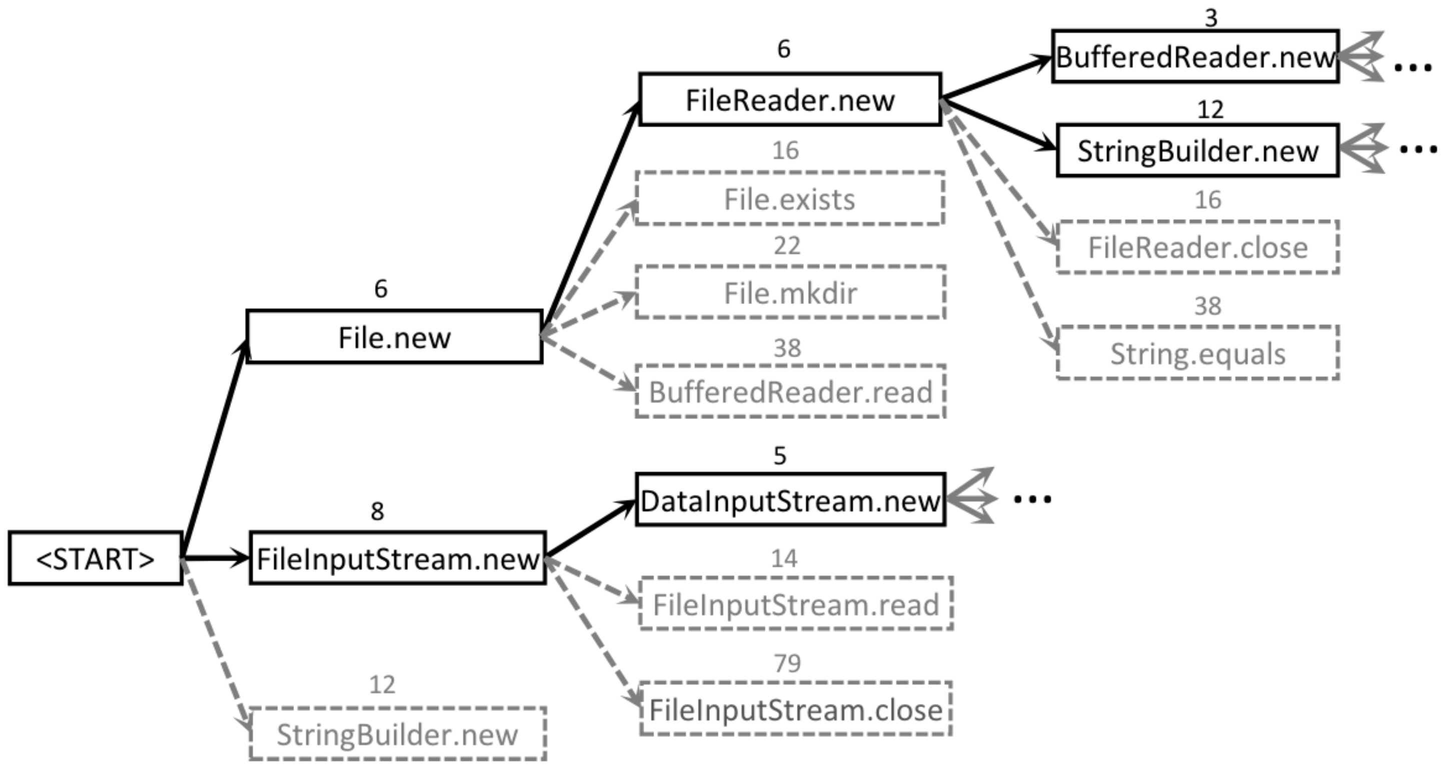
- For a sequence of statements *$stmt_1$; $stmt_2$;...; $stmt_t$*, extract the API sequence s_i from each statement *$stmt_i$*, concatenate them to produce the API sequence s_1-s_2-…-s_t

- For conditional statements such as *if ($stmt_1$) { $stmt_2$; } else { $stmt_3$; }*,create a sequence from all possible branches s_1-s_2-s_3

- For loop statements such as *while($stmt_1$){$stmt_2$;}*, we produce a sequence s_1-s_2

Translation

- Beam search: a heuristic search strategy to find API sequences with the least cost value



$$\text{cost}_{it} = -\log p_\theta(y_{it}|x_i) - \lambda w_{idf}(y_t)$$

Performance Measure

- **Accuracy** of generated API sequences: how close a candidate sequence is to a reference one

- **BLEU score**: the hits of n-grams of a candidate sequence to the reference

$$BLEU = BP \cdot \exp(\sum_{n=1}^{N} w_n \log p_n)$$

Penalize overly short candidates

Weight of each p_n

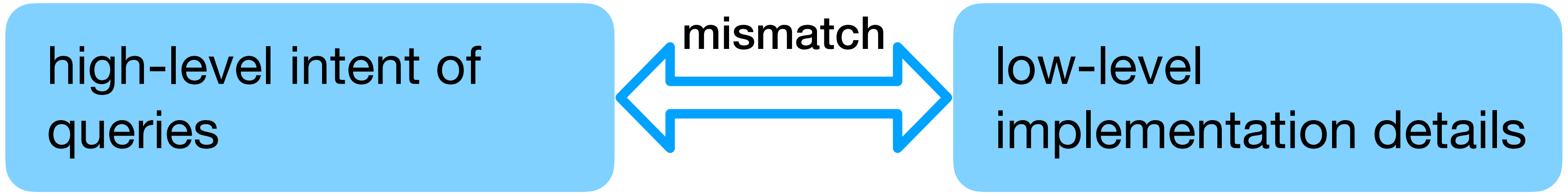
$$p_n = \frac{\#\ \text{n-grams appear in the reference}+1}{\#\ \text{n-grams of candidate}+1} \quad \text{for } n = 1, ..., N$$

- Human evaluation: FRank and relevancy ratio

Tool	Top1	Top5	Top10
Lucene+UP-Miner	11.97	24.08	29.64
SWIM	19.90	25.98	28.85
DeepAPI	54.42	64.89	67.83

A Harder Task: Code Search

- Reuse previously written code snippets by searching through code base
- A harder task: the semantics of code snippets are related to not only the API sequences but also tokens and method names
- Conventional methods: information retrieval based code search, keyword matching of method signatures, text similarity and API matching

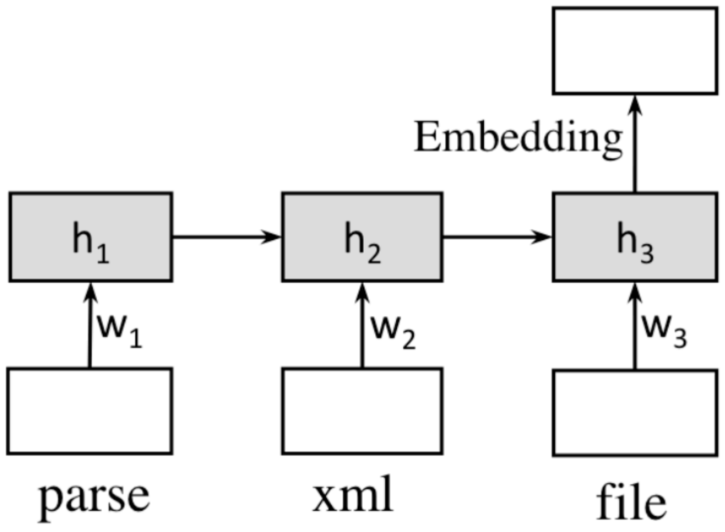

high-level intent of queries
mismatch
low-level implementation details

RNN for Sequence Embedding

- Each word is first mapped to a d-dimensional vector w_t by word embedding. Then the hidden state h_t is updated at time t by

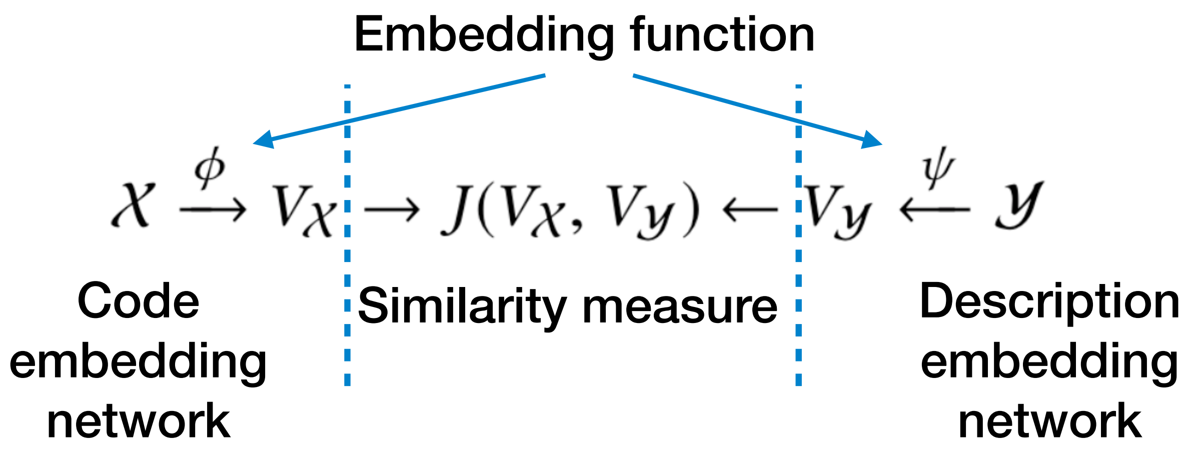
$$\boldsymbol{h}_t = \tanh(\boldsymbol{W}\,[\boldsymbol{h}_{t-1}; \boldsymbol{w}_t]),\ \forall t = 1, 2, \ldots, T$$

- Sentence embedding is summarized from the hidden states $h_1, \ldots, h_T$



Joint Embedding of Hetero Data

- Jointly embed X and Y into a unified vector space so that semantically similar concepts are aligned



$$\mathcal{X} \xrightarrow{\phi} V_{\mathcal{X}} \rightarrow J(V_{\mathcal{X}}, V_{\mathcal{Y}}) \leftarrow V_{\mathcal{Y}} \xleftarrow{\psi} \mathcal{Y}$$

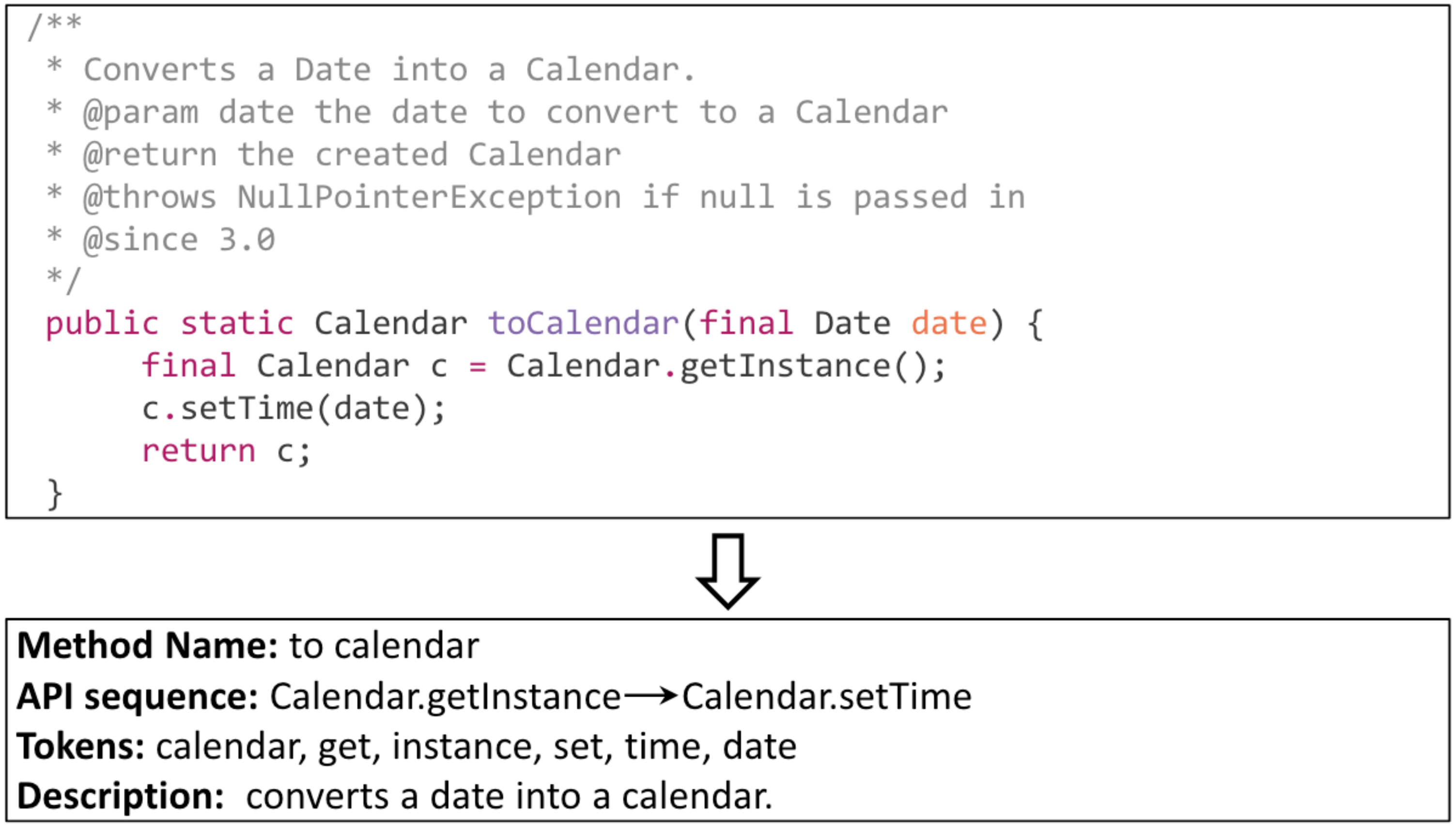
- Code embedding network: embeds source code, including method name, API invocation sequence, and tokens
- Description embedding network: embeds natural language descriptions

Collecting Corpus

- Extract <method name, API sequence, tokens, description> tuples
- **Method name extraction**: extract its name and parse the name into a sequence of tokens. E.g., *listFiles* is parsed into the tokens *list* and *files*
- **API sequence extraction**: similar to API learning
- **Token Extraction**: tokenize the method body, split each token according to camel case, remove the duplicated tokens, remove stop words (such as the and in) and keywords (not discriminative)
- **Description Extraction**: parse AST from code, extract the documentation from the AST

Collecting Corpus

- Extracting code elements from a Java method



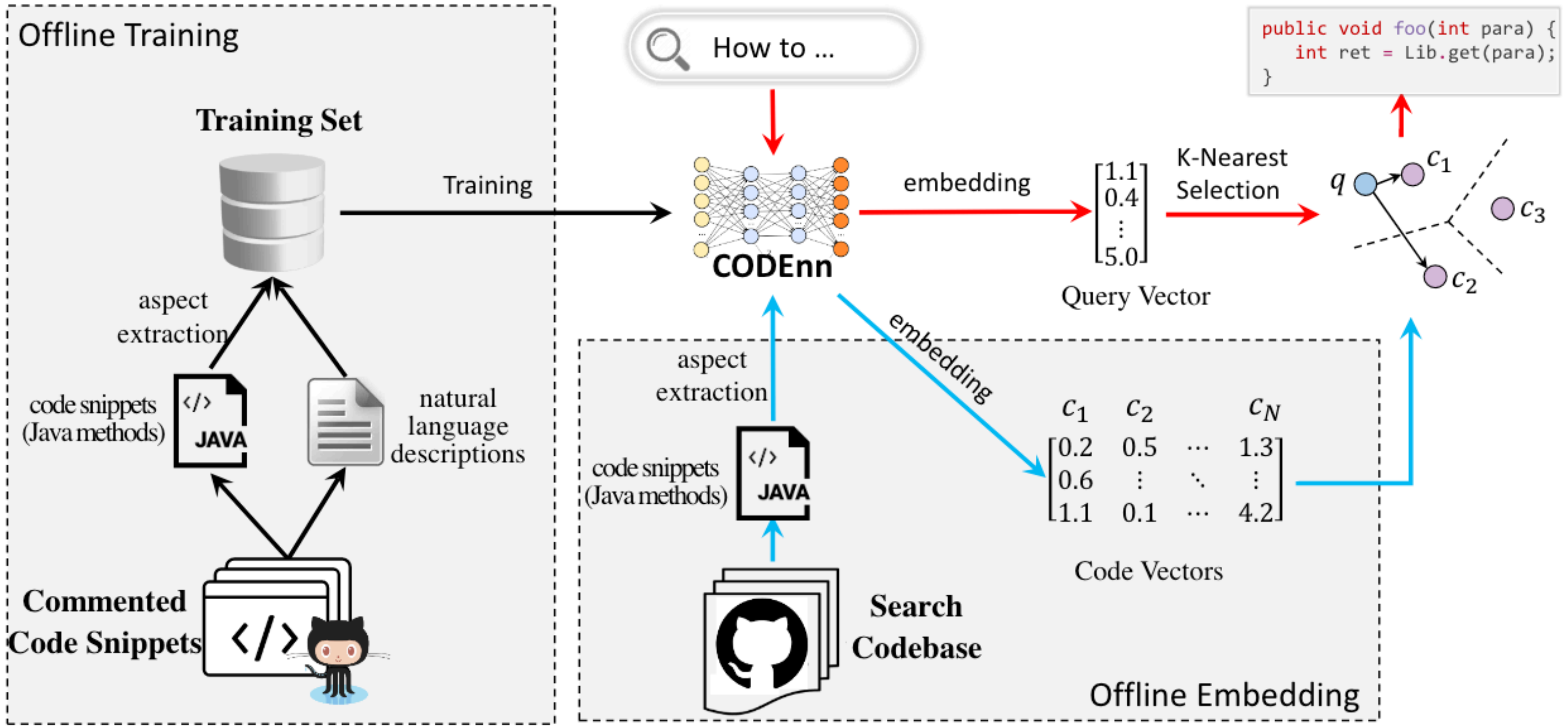
```java
/**
 * Converts a Date into a Calendar.
 * @param date the date to convert to a Calendar
 * @return the created Calendar
 * @throws NullPointerException if null is passed in
 * @since 3.0
 */
public static Calendar toCalendar(final Date date) {
     final Calendar c = Calendar.getInstance();
     c.setTime(date);
     return c;
}
```

$\Downarrow$

Method Name: to calendar
API sequence: Calendar.getInstance⟶Calendar.setTime
Tokens: calendar, get, instance, set, time, date
Description: converts a date into a calendar.

Model Training

Code similarity:
cos(code, description)



$$\begin{bmatrix} 1.1 \\ 0.4 \\ \vdots \\ 5.0 \end{bmatrix}$$

$$\begin{array}{cccc} c_1 & c_2 & & c_N \\ \end{array}$$
$$\begin{bmatrix} 0.2 & 0.5 & \cdots & 1.3 \\ 0.6 & \vdots & \ddots & \vdots \\ 1.1 & 0.1 & \cdots & 4.2 \end{bmatrix}$$

```
public void foo(int para) {
   int ret = Lib.get(para);
}
```

Search: estimates the cos sim between query vector and all code vectors, and top K are selected

Experiment

- Select 16M methods from starred projects in GitHub
- Build a benchmark of queries from top 50 voted Java programming questions in Stack Overflow
- Performance measure: FRank (rank of the first hit result), SuccessRate@k, Precision@k (percentage of relevant results in top k returned results), MRR (inverse of FRank)

```java
public static < S > S deserialize(Class c, File xml) {
    try {
        JAXBContext context = JAXBContext.newInstance(c);
        Unmarshaller unmarshaller = context.createUnmarshaller();
        S deserialized = (S) unmarshaller.unmarshal(xml);
        return deserialized;
    } catch (JAXBException ex) {
        log.error("Error-deserializing-object-from-XML", ex);
        return null;
    }
}
```

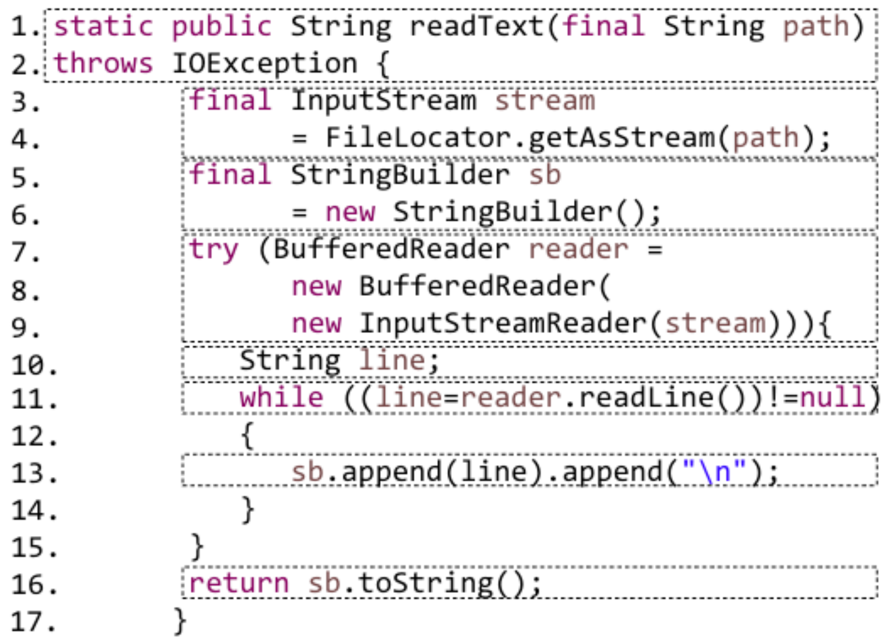
The 1st result of the query “read an object from an xml file”

- What are other learning approaches for source code?

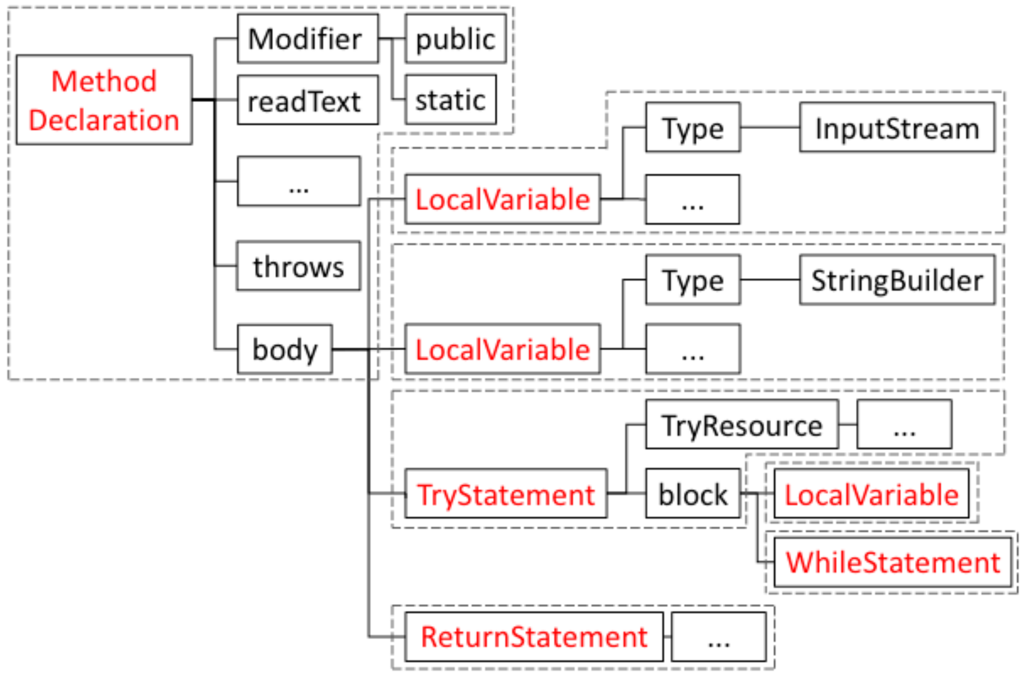
Programs as Trees

- Abstract syntax trees are usually very large
- **ASTNN** splits a large AST into a sequence of small statement trees, encodes the trees to vectors
- One tree consists of AST nodes of one statement and rooted at the Statement node
  - Design a recursive encoder on multi-way statement trees to capture the statement-level lexical and syntactical info
  - Use bidirectional Gated Recurrent Unit to obtain vector representation of the code fragment

Statement Trees



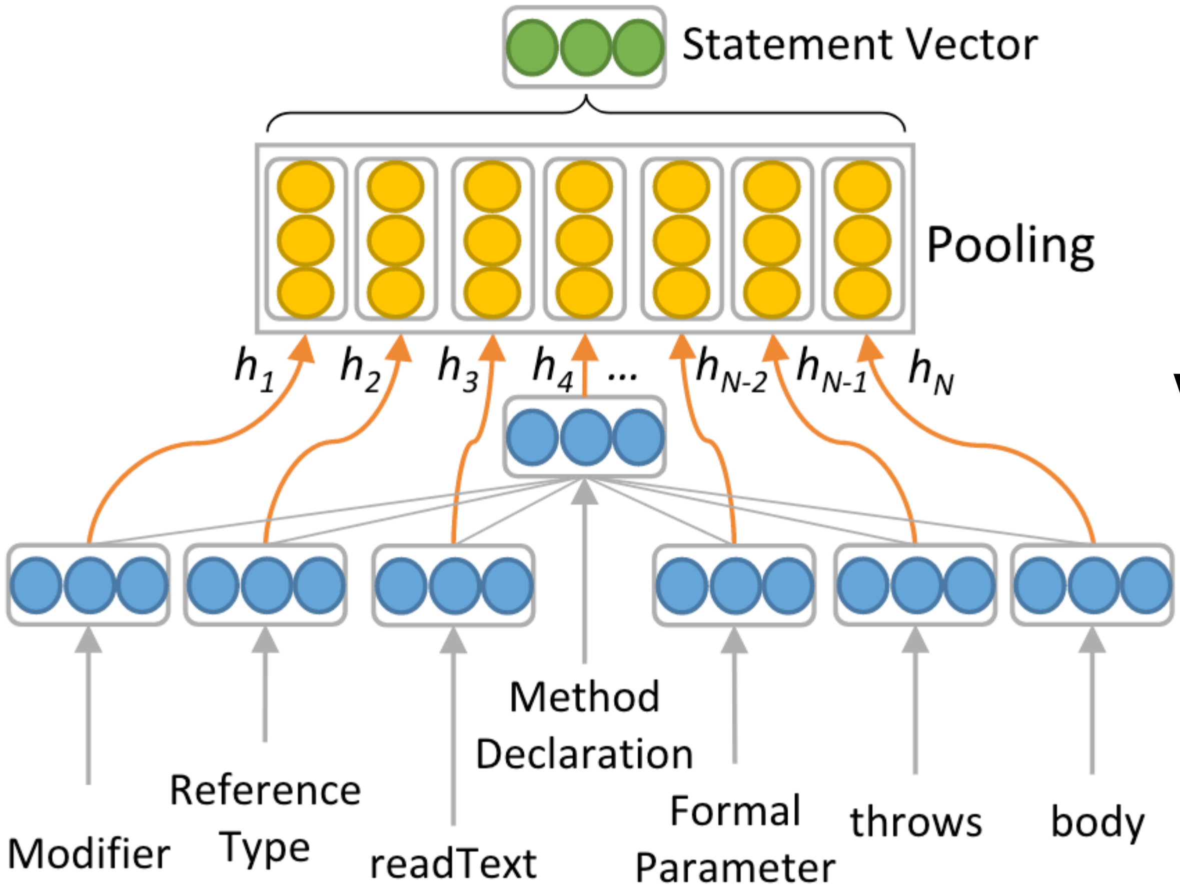
```
static public String readText(final String path)
throws IOException {
        final InputStream stream
                = FileLocator.getAsStream(path);
        final StringBuilder sb
                = new StringBuilder();
        try (BufferedReader reader =
                new BufferedReader(
                new InputStreamReader(stream))){
            String line;
            while ((line=reader.readLine())!=null)
            {
                sb.append(line).append("\n");
            }
        }
        return sb.toString();
       }
```



Code fragments and statements

AST and statement trees

Encoding ST-trees



Lexical vector of node n
X_n is one-hot representation

$$v_n = {W_e}^{\top} x_n$$

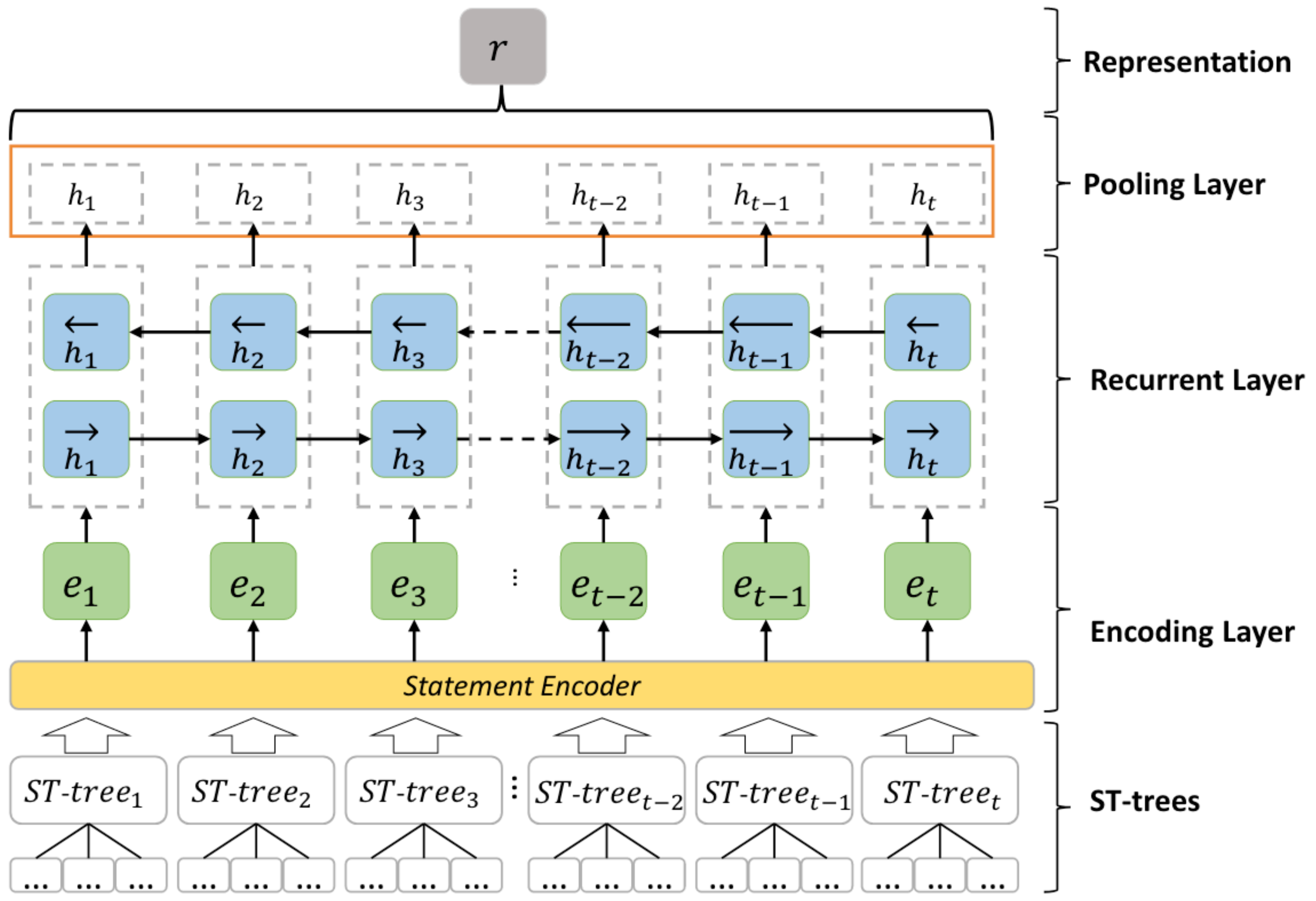
Vector representation of node n

$$h = \sigma({W_n}^{\top} v_n + \sum_{i \in [1,C]} h_i + b_n)$$

Nodes are sampled by
max pooling

$$e_t = [max(h_{i1}), \cdots, max(h_{ik})]$$

AST-based Neural Network



r
Representation
h_1 h_2 h_3 h_{t-2} h_{t-1} h_t
Pooling Layer
$\overleftarrow{h_1}$ $\overleftarrow{h_2}$ $\overleftarrow{h_3}$ $\overleftarrow{h_{t-2}}$ $\overleftarrow{h_{t-1}}$ $\overleftarrow{h_t}$
$\overrightarrow{h_1}$ $\overrightarrow{h_2}$ $\overrightarrow{h_3}$ $\overrightarrow{h_{t-2}}$ $\overrightarrow{h_{t-1}}$ $\overrightarrow{h_t}$
Recurrent Layer
e_1 e_2 e_3 e_{t-2} e_{t-1} e_t
Statement Encoder
Encoding Layer
$ST\text{-}tree_1$ $ST\text{-}tree_2$ $ST\text{-}tree_3$ $ST\text{-}tree_{t-2}$ $ST\text{-}tree_{t-1}$ $ST\text{-}tree_t$
...
ST-trees

Applications of ASTNN

- **Source code classification**: classify code fragments by functionalities

Train loss:

$$J(\Theta, \hat{x}, y) = \sum \left(-log \frac{exp(\hat{x}_y)}{\sum_j exp(\hat{x}_j)} \right)$$

$\hat{x}$**: logit of code fragment vector**

y**: true label**

- **Code clone detection**: detect whether two code fragments implement the same functionality

Train loss:

$$J(\Theta, \hat{y}, y) = \sum (-(y \cdot log(\hat{y}) + (1 - y) \cdot log(1 - \hat{y})))$$

$$\hat{y} = sigmoid(\hat{x})$$

$\hat{x}$ **represents semantic relatedness**

Programs as Graphs

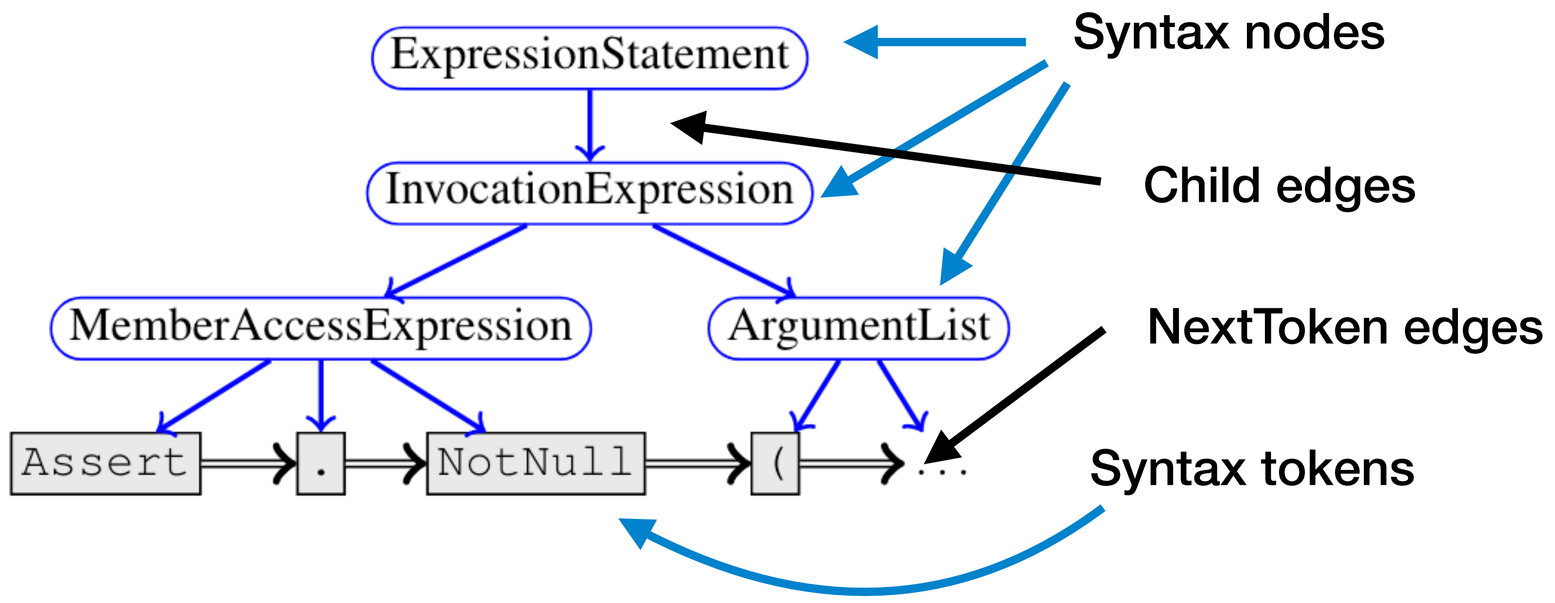
- Drawback of sequence representation: long-range dependencies induced by using the same variable or function in distant locations are not considered
- Use graphs to represent structure of code
- Encode programs as graphs, in which edge represent **syntactic relationships** (e.g., “token before/after”) as well as **semantic relationships** (“variable last used/written here”)
    - Semantic role of a variable (e.g., “is it a counter?”, “is it a filename?”)
    - Take advantage of data flow information

Programs as Graphs

- Gated graph neural networks G = (V, E, X), where V: nodes, E: edges, X: node features
- Backbone: abstract syntax tree, consisting of syntax nodes (non-terminals) and syntax tokens (terminals)
- Different edge types model **syntactic** and **semantic** relations between different tokens
  - Use *Child* edges to connect nodes according to AST
  - Add *NextToken* edges connecting each syntax token to its successor
  - Add edges connecting different uses and updates of syntax tokens corresponding to variables (*LastRead*, *LastWrite*)
  - For v = expr, connect v to all var tokens occurring in expr using *ComputedFrom* edges

An Example

```
Assert.NotNull(clazz);
```



Another Example

- Data flow edges for

```
( x, y) = Foo( );

while ( x > 0 ) x = x + y
```

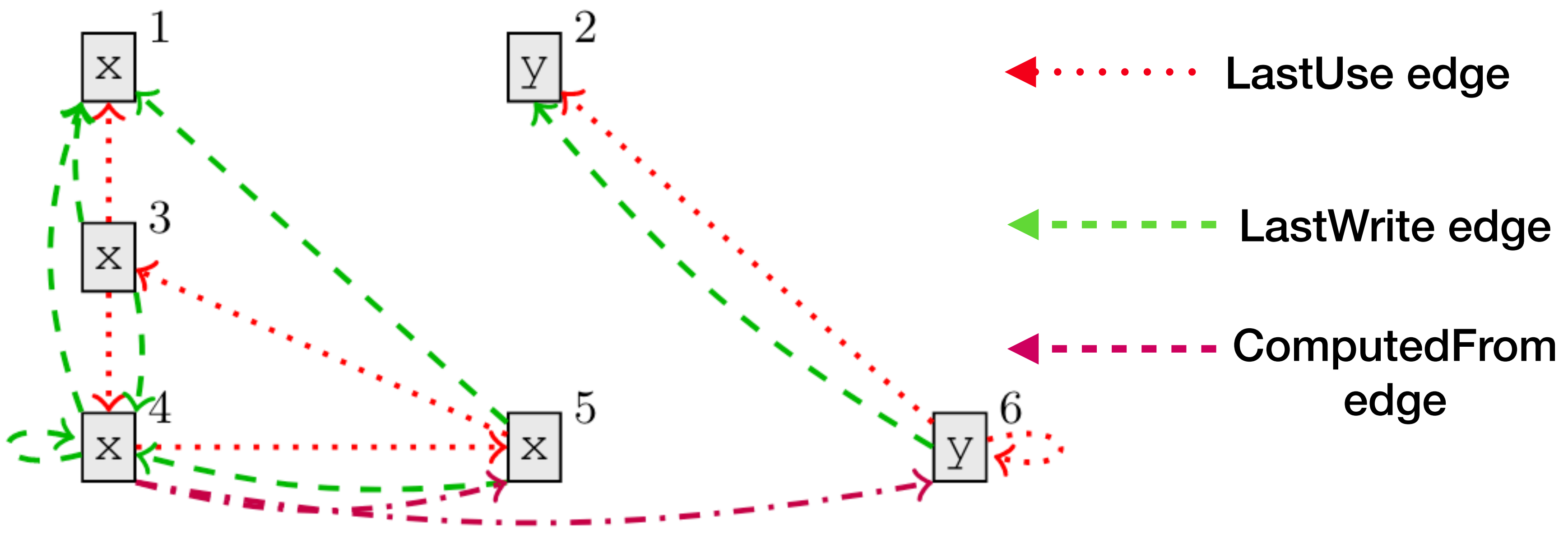
Superscripts are to distinguish different appearance of the same var

Another Example

- Data flow edges for

```
( x¹, y²) = Foo( );

while ( x³ > 0 ) x⁴ = x⁵ + y⁶
```

Superscripts are to distinguish different appearance of the same var



Node Representation

- Combine **textual representation** and its **type**
- Split name of a node into **subtokens**, average the embeddings of all subtokens to retrieve an embedding for the node name
- Associate node *v* with a state vector *h*, initialized from node label *x*, **propagate “messages”** from each *v* to its **neighbors**, where each message is computed from its current state *m = f(h)*

User Case: VarMisuse

- Detect variable misuses: need to infer the role and function of the program elements
- View a code file as a sequence of tokens t_0 … t_N = T. For each slot t_m, given t_0, …, t_{m-1} and t_{m+1}, …, t_N, select t_m from all type-correct variables (w/o raising a compiler error)
- E.g., detect the marked *clazz* is a mistake, *first* should be used instead (C#)

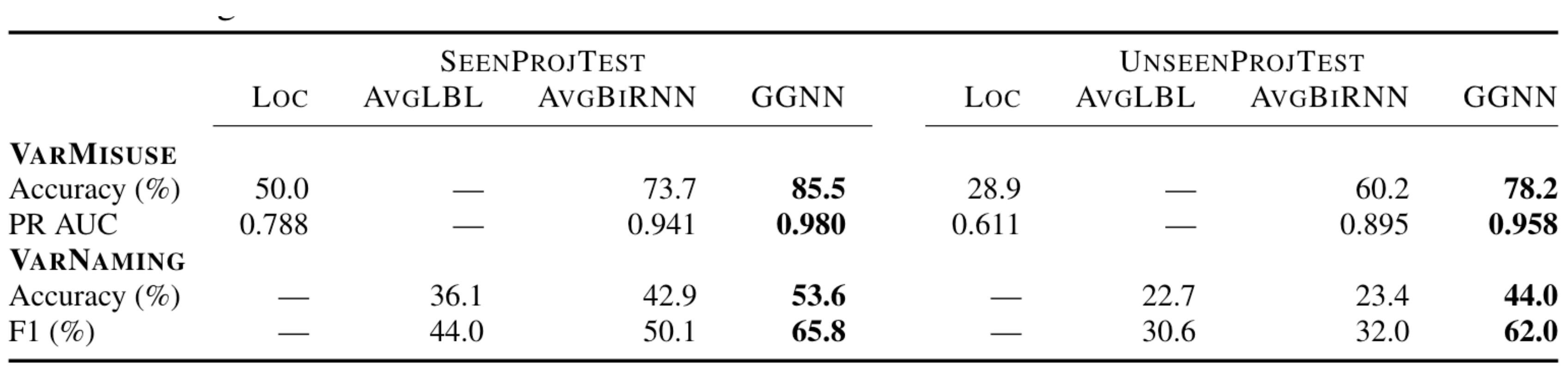
```
var clazz=classTypes["Root"].Single() as JsonCodeGenerator.ClassType;
Assert.NotNull(clazz);

var first=classTypes["RecClass"].Single() as JsonCodeGenerator.ClassType;
Assert.NotNull( clazz );

Assert.Equal("string", first.Properties["Name"].Name);
Assert.False(clazz.Properties["Name"].IsArray);
```

VarMisuse

- Compute a context representation for the slot where we want to predict the used variable, connect it to the remaining graph using all applicable edges that do not depend on the chosen variable
- Compute the usage representation of each candidate variable v at the target slot, connect it to the graph by inserting edges
- Correct variable usage at the location is computed as
  $\arg\max_v W[\textit{context representation}, \textit{usage representation}]$

Experiment

- Baselines: bidirectional RNN-based baselines LoC, AvgBiRNN
    - LoC runs over the tokens before and after the target location
    - AbgBiRNN has an additional usage representation run over the token before/after each usage
- LoC captures very little information, AvgBiRNN captures information from many variable usage sites, but do not explicitly encode the rich structure of the problem



	SeenProjTest				UnseenProjTest			
	Loc	AvgLBL	AvgBiRNN	GGNN	Loc	AvgLBL	AvgBiRNN	GGNN
VarMisuse								
Accuracy (%)	50.0	—	73.7	**85.5**	28.9	—	60.2	**78.2**
PR AUC	0.788	—	0.941	**0.980**	0.611	—	0.895	**0.958**
VarNaming								
Accuracy (%)	—	36.1	42.9	**53.6**	—	22.7	23.4	**44.0**
F1 (%)	—	44.0	50.1	**65.8**	—	30.6	32.0	**62.0**

Reading

- Gu, Xiaodong, et al. "Deep API learning." Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering. 2016.
- Gu, Xiaodong, Hongyu Zhang, and Sunghun Kim. "Deep code search." Proceedings of the 40th International Conference on Software Engineering. 2018.
- Zhang, Jian, et al. "A novel neural source code representation based on abstract syntax tree." 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019.
- Allamanis, Miltiadis, Marc Brockschmidt, and Mahmoud Khademi. "Learning to Represent Programs with Graphs." International Conference on Learning Representations. 2018