CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation

Yue Wang[1], *Weishi Wang*[1,2], *Shafiq Joty*[1,2], *Steven Hoi*[1]

[1]Salesforce Research Asia [2]Nanyang Technological University




Motivation

Inspired by the success of pre-trained language models in NLP, the recent years witness the surge of **pre-trained programming language models**, e.g., CodeBERT and CodeGPT

However, existing programming language models have **two shortcomings**:

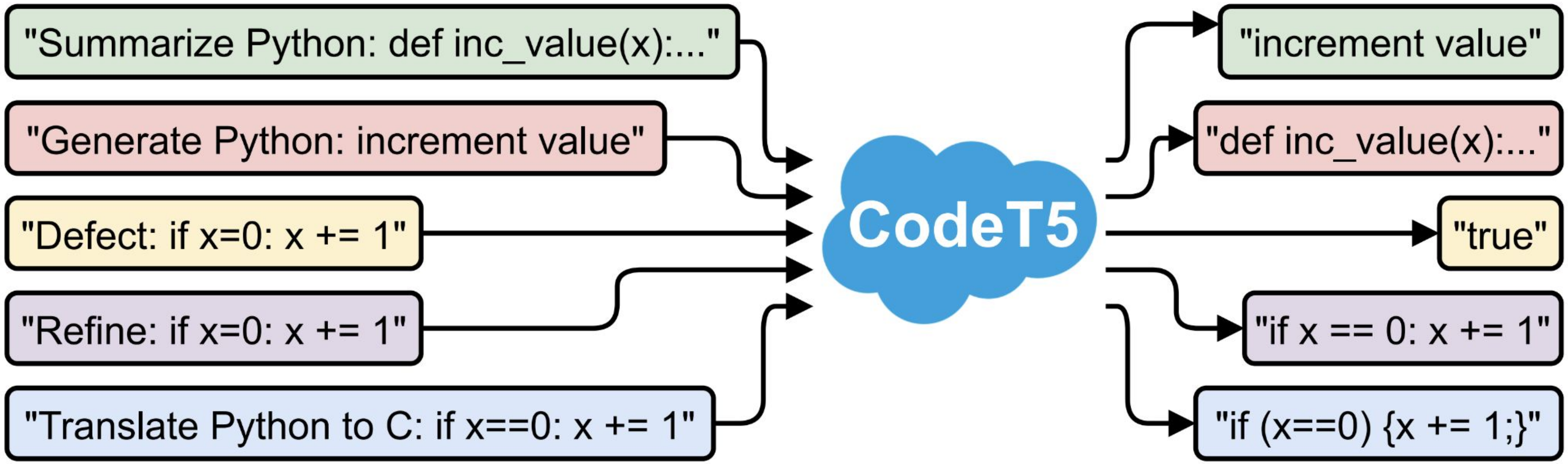
- Most current methods either rely on an encoder-only (or decoder-only) pre-training that is suboptimal for generation (resp. understanding) tasks
- They often process the code snippet in the same way as natural language (NL), neglecting the special characteristics of programming language (PL) such as code token types

Can we propose a unified model to support all tasks of both types?

Can we leverage more code-specific knowledge?

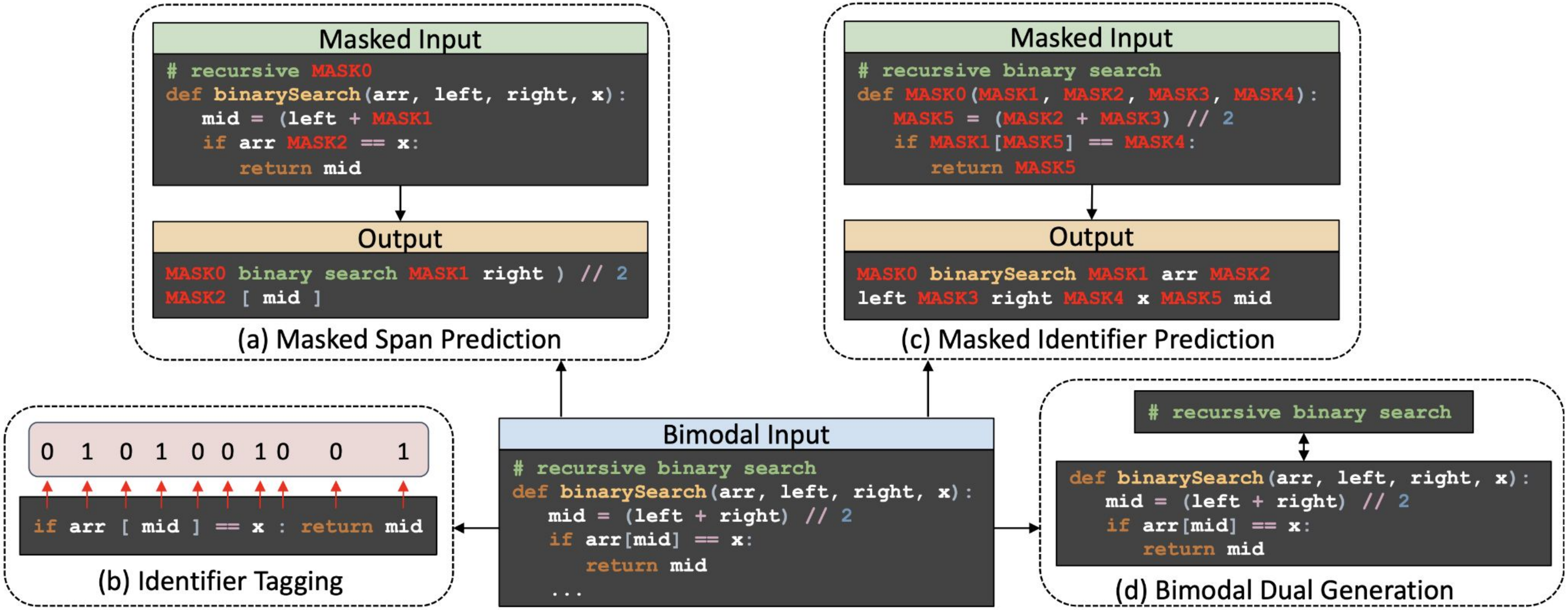
Contributions

- Present CodeT5, a novel pre-trained encoder-decoder model that supports both understanding and generation tasks and also allows for multi-task learning
- Propose an *identifier-aware denoising objective* to fuse the **code token type** information and a *bimodal dual generation task* to learn a better NL-PL alignment
- CodeT5 yields new state-of-the-art results on **fourteen sub-tasks** in CodeXGLUE benchmark



Pre-training Tasks of CodeT5

Overview



Pre-training Tasks of CodeT5

Two-stage pre-training

Bimodal Input

```
# recursive binary search
def binarySearch(arr, left, right, x):
   mid = (left + right) // 2
   if arr[mid] == x:
      return mid
   ...
```

Input **x** = ([CLS], w_1, ..., w□, [SEP], c_1, ..., c□, [SEP])

NL words　　PL code tokens

S1: Identifier-aware denoising objective

- Masked Span Prediction (MSP)
- Identifier Tagging (IT)
- Masked Identifier Prediction (MIP)

} Bimodal input or unimodal input

S2: Bimodal Dual Generation

- Dual conversion between NL and PL

} Bimodal input only

Pre-training Tasks of CodeT5

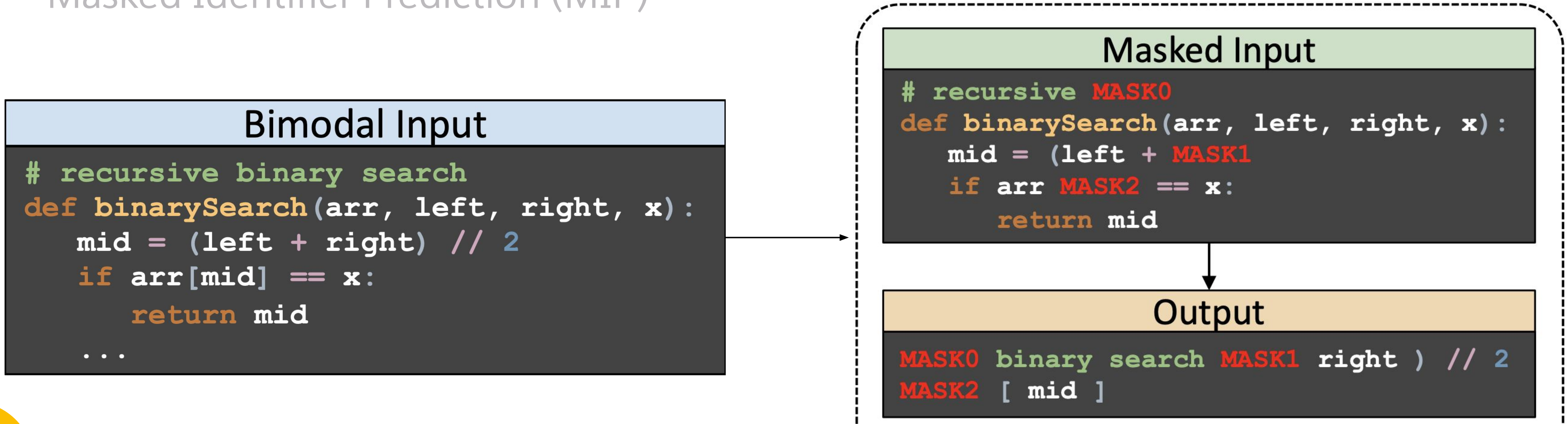
Two-stage pre-training

S1: Identifier-aware denoising objective

- Masked Span Prediction (MSP)
- Identifier Tagging (IT)
- Masked Identifier Prediction (MIP)

Similar to default T5 objective, but differs in

- Whole word masking
- Denoising on NL-PL bimodal input



(a) Masked Span Prediction

Pre-training Tasks of CodeT5

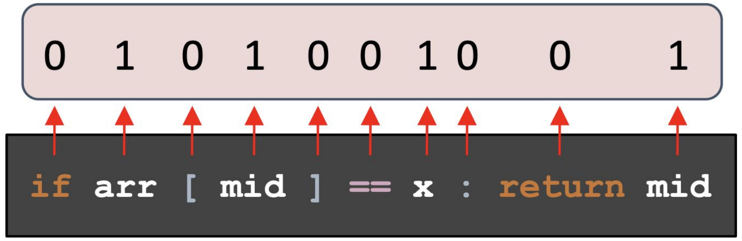
Two-stage pre-training

S1: Identifier-aware denoising objective

- Masked Span Prediction (MSP)
- Identifier Tagging (IT)
  - **Syntax highlight**: to distinguish which code tokens are identifiers
- Masked Identifier Prediction (MIP)

Bimodal Input

```
# recursive binary search
def binarySearch(arr, left, right, x):
    mid = (left + right) // 2
    if arr[mid] == x:
        return mid
    ...
```



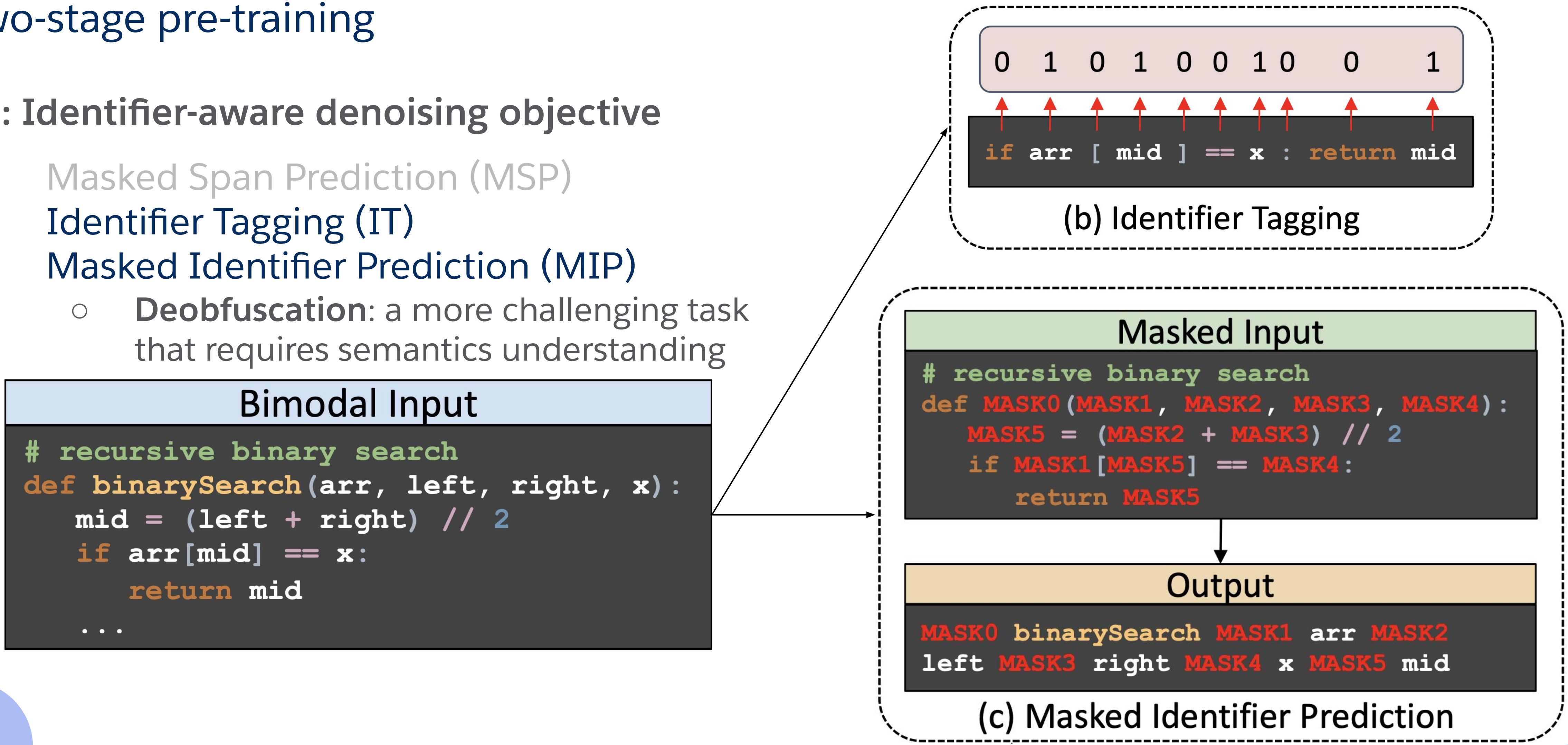
(b) Identifier Tagging

Pre-training Tasks of CodeT5

Two-stage pre-training

S1: Identifier-aware denoising objective

- Masked Span Prediction (MSP)
- Identifier Tagging (IT)
- Masked Identifier Prediction (MIP)
  - **Deobfuscation**: a more challenging task that requires semantics understanding

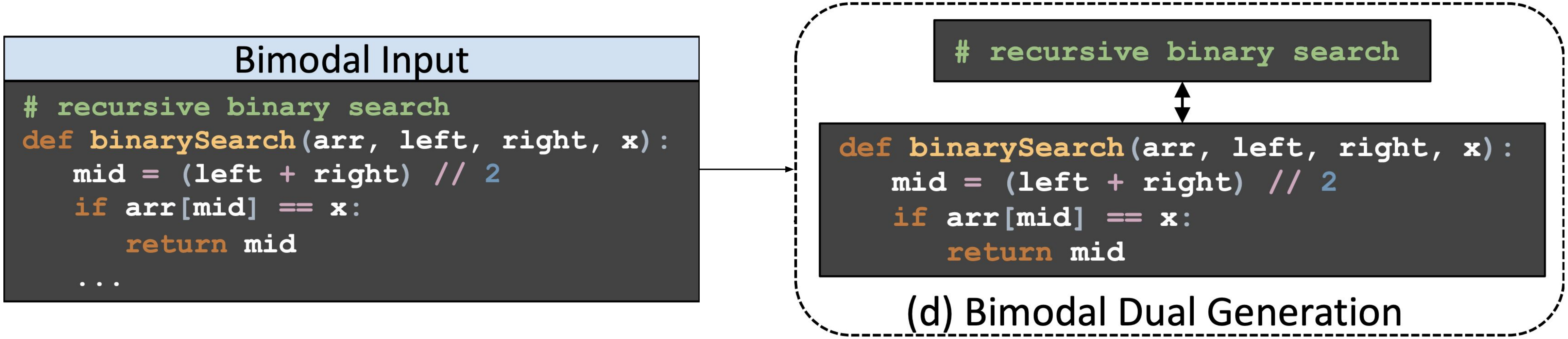


Pre-training Tasks of CodeT5

Two-stage pre-training

S2: Bimodal Dual Generation

- Dual conversion either from NL to PL or from PL to NL
    - Leverage the NL-PL pairs (function and its comments) naturally available in source code to learn a **better cross-modal alignment**



Pre-training Dataset & Tokenizer

Pre-training dataset

- Size: 8.35M (3.16M bimodal+5.19M unimodal)
  - 6 PLs from CodeSearchNet
  - 2 PLs (C/C#) from BigQuery
- Use tree-sitter to parse the function into an abstract syntax tree (AST) and extract identifiers
- Identifier rate: 19.32%~32.08%

	PLs	W/ NL	W/o NL	Identifier
CodeSearchNet	Ruby	49,009	110,551	32.08%
	JavaScript	125,166	1,717,933	19.82%
	Go	319,132	379,103	19.32%
	Python	453,772	657,030	30.02%
	Java	457,381	1,070,271	25.76%
	PHP	525,357	398,058	23.44%
Our	C	1M	-	24.94%
	CSharp	228,496	856,375	27.85%
	Total	3,158,313	5,189,321	8,347,634

Table 1: Dataset statistics. "Identifier" denotes the proportion of identifiers over all code tokens for each PL.

Code-specific tokenizer

- Build our own byte-level BPE tokenizer using our training data (vocabulary=32K)
- Default T5 tokenizer encodes some common code tokens (e.g., '{' and '}') into <unk>
- Ours reduce the tokenized code length (**30%~45%**) ⇒ accelerate training!

Fine-tuning on Downstream Tasks

Task-specific transfer learning: fine-tune on each of tasks in CodeXGLUE benchmark

Generation tasks

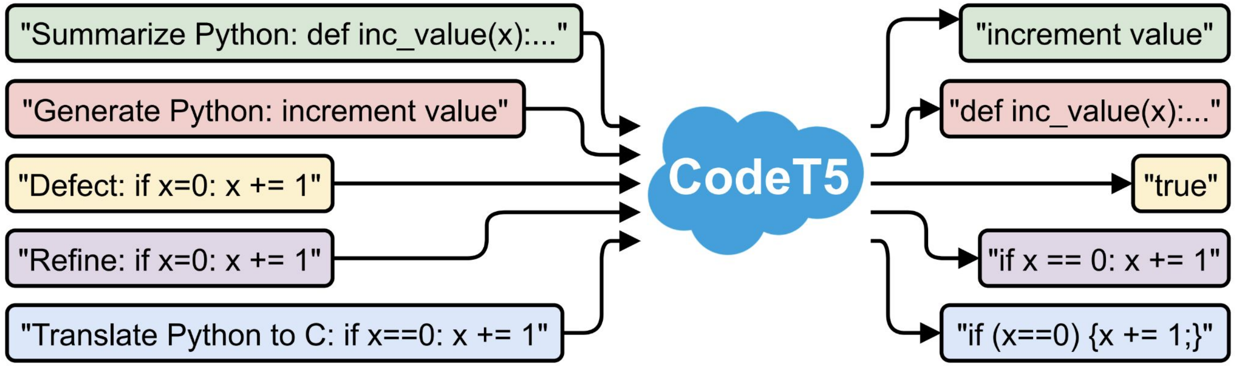
- Summarization (PL→ NL)
- Generation (NL→ PL)
- Refinement (buggy PL → correct PL)
- Translation (PL 1 → PL 2)

Understanding tasks

- Defect detection (PL → 0/1)
- Clone detection (PL 1 + PL 2 → 0/1)

Multi-task learning

- Employ a unified set of task control prompts, e.g., “Translate Python to C”
- Balanced sampling & allow to select different checkpoints for different tasks



Experiment Results

NL↔PL: code summarization & generation

Observations:

- Both CodeT5-small and CodeT5-base are much better than the SOTA PLBART
- Dual-gen benefits both NL↔PL tasks while multi-task only benefits the summarization task

Methods	Ruby	JavaScript	Go	Python	Java	PHP	Overall
RoBERTa	11.17	11.90	17.72	18.14	16.47	24.02	16.57
CodeBERT	12.16	14.90	18.07	19.06	17.65	25.16	17.83
DOBF	-	-	-	18.24	19.05	-	-
PLBART	14.11	15.56	18.91	19.30	18.45	23.58	18.32
CodeT5-small	14.87	15.32	19.25	20.04	19.92	25.46	19.14
+dual-gen	15.30	15.61	19.74	19.94	19.78	26.48	19.48
+multi-task	15.50	15.52	19.62	20.10	19.59	25.69	19.37
CodeT5-base	15.24	16.16	19.56	20.01	20.31	26.03	19.55
+dual-gen	**15.73**	16.00	19.71	20.11	20.41	**26.53**	19.75
+multi-task	15.69	**16.24**	**19.76**	**20.36**	**20.46**	26.09	**19.77**

Table 2: Smoothed BLEU-4 scores on the code summarization task. The "Overall" column shows the average scores over six PLs. Best results are in bold.

Methods	EM	BLEU	CodeBLEU
GPT-2	17.35	25.37	29.69
CodeGPT-2	18.25	28.69	32.71
CodeGPT-adapted	20.10	32.79	35.98
PLBART	18.75	36.69	38.52
CodeT5-small	21.55	38.13	41.39
+dual-gen	19.95	39.02	42.21
+multi-task	20.15	35.89	38.83
CodeT5-base	22.30	40.73	43.20
+dual-gen	**22.70**	**41.48**	**44.10**
+multi-task	21.15	37.54	40.01

Table 3: Results on the code generation task. EM denotes the exact match.

Experiment Results

PL→ PL: code translation & refinement

Code translation (Java-C#)

- CodeT5-base is consistently better than PLBART while -small is comparable
- Both dual-gen and multi-task do not help, while dual-gen even hurts

Code refinement (Java small/medium)

- Exact match (EM) is more important
- CodeT5-small and -base achieve SOTA results, especially on the medium set
- Dual-gen still hurts while multi-task significantly boosts the performance

Methods	Java to C#		C# to Java		Refine Small		Refine Medium	
	BLEU	EM	BLEU	EM	BLEU	EM	BLEU	EM
Naive Copy	18.54	0	18.69	0	78.06	0	90.91	0
RoBERTa (code)	77.46	56.10	71.99	57.90	77.30	15.90	90.07	4.10
CodeBERT	79.92	59.00	72.14	58.80	77.42	16.40	91.07	5.20
GraphCodeBERT	80.58	59.40	72.64	58.80	**80.02**	17.30	**91.31**	9.10
PLBART	83.02	64.60	78.35	65.00	77.02	19.21	88.50	8.98
CodeT5-small	82.98	64.10	79.10	65.60	76.23	19.06	89.20	10.92
+dual-gen	82.24	63.20	78.10	63.40	77.03	17.50	88.99	10.28
+multi-task	83.49	64.30	78.56	65.40	77.03	20.94	87.51	11.11
CodeT5-base	**84.03**	**65.90**	**79.87**	**66.90**	77.43	21.61	87.64	13.96
+dual-gen	81.84	62.00	77.83	63.20	77.66	19.43	90.43	11.69
+multi-task	82.31	63.40	78.01	64.00	78.06	**22.59**	88.90	**14.18**

Table 4: BLEU-4 scores and exact match (EM) accuracies for code translation (Java to C# and C# to Java) and code refinement (small and medium) tasks.

Experiment Results

Understanding tasks: code defect/clone detection

Metric: accuracy for code defect detection and F1 score for code clone detection

Observations:

- CodeT5 models yield much better accuracy on code defect detection and comparable F1 score on code clone detection
- Bimodal dual generation and multi-task learning do not help and even sometimes hurt

Methods	Defect Accuracy	Clone F1
RoBERTa	61.05	94.9
CodeBERT	62.08	96.5
DOBF	-	96.5
GraphCodeBERT	-	97.1
PLBART	63.18	**97.2**
CodeT5-small	63.40	97.1
+dual-gen	63.47	97.0
+multi-task	63.58	-
CodeT5-base	**65.78**	**97.2**
+dual-gen	62.88	97.0
+multi-task	65.02	-

Table 5: Results on the code defect detection and clone detection tasks.

Ablation Study

Analyzing identifier-aware denoising objective

Representative tasks:

- Generation: PL → NL, NL→ PL, PL→ PL
- Understanding: defect detection

Observations:

- All components contribute to the better overall performance for all tasks
- Masked span prediction (MSP) is crucial for all generation tasks while masked identifier prediction (MIP) is more important for understanding tasks

Methods	Sum-PY (BLEU)	Code-Gen (CodeBLEU)	Refine Small (EM)	Defect (Acc)
CodeT5	20.04	41.39	19.06	63.40
-MSP	18.93	37.44	15.92	64.02
-IT	19.73	39.21	18.65	63.29
-MIP	19.81	38.25	18.32	62.92

Table 6: Ablation study with CodeT5-small on four selected tasks. "Sum-PY" denotes code summarization on Python and "Code-Gen" denotes code generation.

Case Study

Analyzing outputs of CodeT5

- CodeT5 can generate semantically correct output with a better readability
- BLEU is not a perfect metric for evaluating code generation tasks

Type	Code

Target	`public long ramBytesUsed(){`
	`    return BASE_RAM_BYTES_USED+((index!=null)?`
	`        index.ramBytesUsed() : 0);}`

CodeT5	`public long ramBytesUsed(){`
	`    long sizeInBytes = BASE_RAM_BYTES_USED;`
	`    if (index != null){`
	`        sizeInBytes += index.ramBytesUsed();}`
	`    return sizeInBytes;}`

Figure 3: One translation (C# to Java) example that is semantically correct but with a 50.23% BLEU-4 score.

- CodeT5 can produce the correct code snippet while its variant without identifier information fails to do so

Type	Code
Source	`Text: returns the string value of the specified field. the value is obtained from whichever scan contains the field.` `Env: Scan s1; Scan s2; boolean hasField`

CodeT5	`String function (String arg0){`
	`    if (s1.hasField(arg0))`
	`        return s1.getString(arg0);`
	`    else return s2.getString(arg0);}`

W/o MIP+IT	`String function (String arg0){` `    return s1.getString(arg0);}`

Figure 4: One code generation example on Concode test set, where our CodeT5 gives a correct prediction. The important identifiers in the code are highlighted.

Looking Forward

How could CodeT5 disrupt software development?

CodeT5 can be deployed to provide AI-powered coding assistance for software developers

- Text-to-code generation
- Code autocompletion
- Code summarization

```
// convert from one currency to another
public static string convertCurrency(String fromISO, String toISO, String fromCurrency, String toCurrency) {
    if(fromISO == toISO) // no need to convert
        return null;
    else //return the value to the currency
        return UTIL_Currency.getInstance().convertFromISO(fromISO, toISO, fromCurrency);
}


private boolean isContactAddressEmpty(Contact con1)
```

Insert AI Coding Assistant Suggeste… AI Coding Assista…

Looking Forward

How could CodeT5 disrupt software development?

CodeT5 can be deployed to provide AI-powered coding assistance for software developers

- Text-to-code generation
- Code autocompletion
- Code summarization

```
// convert from one currency to another
public static string convertCurrency(String fromISO, String toISO, String fromCurrency, String toCurrency) {
    if(fromISO == toISO) // no need to convert
        return null;
    else //return the value to the currency
        return UTIL_Currency.getInstance().convertFromISO(fromISO, toISO, fromCurrency);
 }


 private boolean isContactAddressEmpty(Contact con1)
{
    boolean isEmpty =
        con1.MailingStreet == null &&
        con1.MailingCity == null &&
        con1.MailingState == null &&
        con1.MailingPostalCode == null &&
        // note that we decided to ignore country so that a default value won't create unnecessary address objects
        (con1.MailingCountry == null || ADDR_Addresses_TDTM.isStateCountryPicklistsEnabled) &&
        // only test StateCode if picklists enabled.
        (!ADDR_Addresses_TDTM.isStateCountryPicklistsEnabled || con1.get('MailingStateCode') == null);
    return isEmpty;
}
```

Conclusion & Ethical Considerations

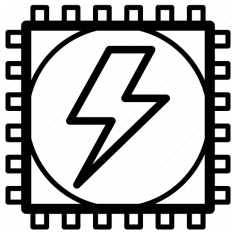
Conclusion

- Present CodeT5, the first code-aware pre-trained encoder-decoder model that yields state-of-the-art results on fourteen sub-tasks in CodeXGLUE benchmark
- A large-scale pre-trained programming language model with great potential to support a wide range of code intelligence applications in the software development lifecycle
- Code and models have been released at github.com/salesforce/CodeT5

Ethical considerations

Dataset Bias

Automation Bias



Computation Cost

Security Implications

Thank You

Find Out More:

Blog: blog.einstein.ai/codet5
Paper: arxiv.org/abs/2109.00859
Code and Models: github.com/salesforce/CodeT5