Towards Making Flowchart Images Machine Interpretable

Shreya Shukla[0009-0002-9029-2547], Prajwal Gatti[0000-0002-6554-3132], Yogesh Kumar[0009-0009-4363-5317], Vikash Yadav[0009-0000-3245-8137], and Anand Mishra[0000-0002-7806-2557]

Vision, Language and Learning Group (VL2G)
IIT Jodhpur, India
{shukla.12, pgatti, kumar.204, yadav.41, mishra}@iitj.ac.in

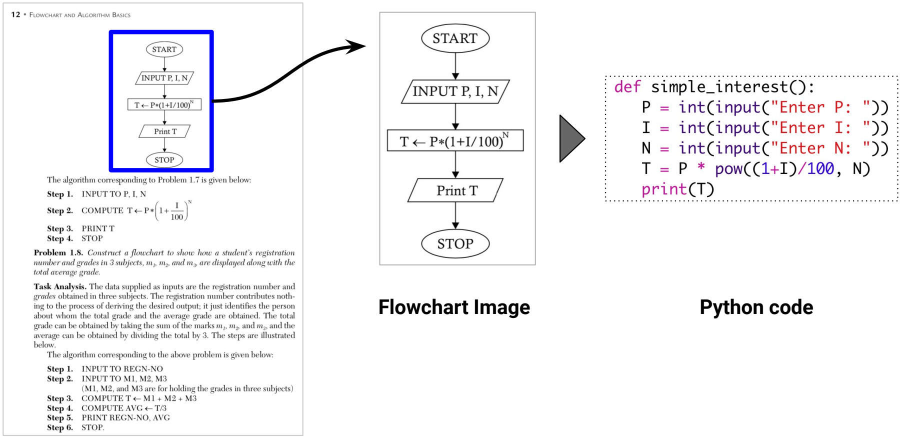
Abstract. Computer programming textbooks and software documentations often contain flowcharts to illustrate the flow of an algorithm or procedure. Modern OCR engines often tag these flowcharts as graphics and ignore them in further processing. In this paper, we work towards making flowchart images machine-interpretable by converting them to executable Python codes. To this end, inspired by the recent success in natural language to code generation literature, we present a novel transformer-based framework, namely FloCo-T5. Our model is well-suited for this task, as it can effectively learn semantics, structure, and patterns of programming languages, which it leverages to generate syntactically correct code. We also used a task-specific pre-training objective to pre-train FloCo-T5 using a large number of logic-preserving augmented code samples. Further, to perform a rigorous study of this problem, we introduce the FloCo dataset that contains 11,884 flowchart images and their corresponding Python codes. Our experiments show promising results, and FloCo-T5 clearly outperforms related competitive baselines on code generation metrics. We make our dataset and implementation publicly available[1].

Keywords: Flowchart Understanding, Code Generation, Large Language Models.

1 Introduction

Flowcharts are widely used across documents to represent algorithms, processes, or workflows in a graphical manner and provide a clear and concise understanding of complex processes. They contain short textual commands or conditions inside various intent-specific shapes, e.g., diamond for decision-making block, rhomboid for input, and output. These shapes are connected with directed or undirected arrows to define a sequential flow of information and processing. In computer programming textbooks and software documentation, flowcharts are more often used as a program-planning tool to communicate the complex logic

[1] https://vl2g.github.io/projects/floco



12 • Flowchart and Algorithm Basics

The algorithm corresponding to Problem 1.7 is given below:

Step 1. INPUT TO P, I, N

Step 2. COMPUTE $T \leftarrow P * \left(1 + \frac{I}{100}\right)^{N}$

Step 3. PRINT T

Step 4. STOP

Problem 1.8. *Construct a flowchart to show how a student's registration number and grades in 3 subjects, m_1, m_2, and m_3, are displayed along with the total average grade.*

Task Analysis. The data supplied as inputs are the registration number and *grades* obtained in three subjects. The registration number contributes nothing to the process of deriving the desired output; it just identifies the person about whom the total grade and the average grade are obtained. The total grade can be obtained by taking the sum of the marks m_1, m_2, and m_3, and the average can be obtained by dividing the total by 3. The steps are illustrated below.

The algorithm corresponding to the above problem is given below:

Step 1. INPUT TO REGN-NO
Step 2. INPUT TO M1, M2, M3
(M1, M2, and M3 are for holding the grades in three subjects)
Step 3. COMPUTE T ← M1 + M2 + M3
Step 4. COMPUTE AVG ← T/3
Step 5. PRINT REGN-NO, AVG
Step 6. STOP.

```python
def simple_interest():
    P = int(input("Enter P: "))
    I = int(input("Enter I: "))
    N = int(input("Enter N: "))
    T = P * pow((1+I)/100, N)
    print(T)
```

Fig. 1. Flow2Code. A scanned document from a programming text book [10] containing a flowchart is shown here. Our aim is to convert flowchart images to executable computer programs. We scope ourselves to cropped flowchart images and Python codes in this work.

of programs and keep track of the data flow through a process, as shown in figure 1. These visual depictions help beginners in programming to focus on formulating the logic behind the program while ignoring the intricacies of the syntax of different programming languages. Machine interpretation of these flowcharts followed by automatic code generation would not only help school students and people from non-software engineering backgrounds but also speed up software development. In order to make these flowchart images machine-interpretable, we study the problem of automatically converting flowchart images to a computer program (code) in a high-level language. This problem is referred to as Flow2Code [16]. Despite its practical importance and utility, Flow2Code has not been rigorously explored in the literature.

There is no existing large-scale dataset in flowchart-to-code literature for performing a rigorous experimental evaluation of Flow2Code task. To fill this gap, we introduce the first dataset, namely FloCo. The FloCo contains 11,884 flowchart images along with corresponding Python codes. Inspired by the success of transformer-based approaches in natural language and code generation tasks [2,3,8,17,26], we present FloCo-T5 – a novel framework to convert flowchart images to the Python code. In our proposed architecture, we first convert the flowchart images into a sequence encoding by automatically detecting different shapes and reading text using off-the-shelf OCR engines [1,20]. Then, to adapt our transformer model to this novel domain, we pre-train it on the masked token modeling objective using a large number of logic-preserving data-augmented code samples. This pre-training step helps the model to understand the structure and semantics of the programming language as well as the flowchart encoding. Finally, we fine-tune the model on train split as a sequence-to-sequence

generation problem where flowchart encoding and expected Python code are used as input and output sequence, respectively. We conducted extensive experiments with the sequence encoding of the flowchart images (as shown in Table 2) and compared the code generation performance of our model against competitive baselines, namely Vanilla Transformer [32], BART [19], PLBART [2] and CodeT5 [33]. Our experiments show that FloCo-T5 outperforms all other baselines on different code generation metrics, showing the efficacy of the proposed pre-training objective and data augmentation techniques. Qualitative results and further analysis (Figures 6 and 7) demonstrate that our model effectively learns the structure and pattern of programming languages and the logical data flow and generates syntactically correct code corresponding to the flowchart images.
Contributions: The major contributions of this work are three folds:

1. We study the Flow2Code task in a "large-scale setting" and introduce an accompanying dataset – FloCo containing 11,884 flowchart images and corresponding Python codes. This dataset shall enable future research in this under-explored space (Section 3).
2. We propose a novel framework viz. FloCo-T5 to address the task in hand, which involves generating flowchart encodings, pre-training CodeT5 on the task-specific objective with augmented codes, and finally fine-tuning for the code generation task (Section 4).
3. We conducted extensive experiments with various baselines and proposed task-specific code augmentation and pre-training strategy. We achieve BLEU, CodeBLEU, and exact match scores of 67.4, 75.7, and 20.0, respectively. Towards the end, we show that our model can be adopted to hand-drawn flowchart images as well (Section 5).

2 Related Work

2.1 Flowchart Understanding

There have been several attempts to build software for flowchart-to-code conversion, such as authors in [12], and [30] introduced interactive user interfaces to convert flowcharts to codes on-the-fly in various programming languages. These rule-based approaches, however, impose restrictions and do not support the conversion for offline flowchart images like ours. In [35], a platform was designed to recognize flowcharts and convert them to ANSI-C code using structure identification. In [16], a method was proposed for handwritten flowcharts, using rule-based techniques for preprocessing and generating pseudo code. In [9], improved results were achieved in flowchart recognition by combining statistical and structural information. In [28], the Faster RCNN object detection system was extended with an arrow keypoint predictor to recognize handwritten flowcharts. In [13], DrawnNet was proposed, a keypoint-based detector for handwritten diagram recognition.

A recent work [31] introduced a novel benchmark and dataset for question-answering over flowcharts. However, their flowchart images are unsuited for programming tasks and can not be used for our problem. The work closest to our setting is [23], which targets the digitization of handwritten flowchart images with Faster RCNN and OCR-techniques, followed by converting them to codes in C programming language using a CNN-LSTM based model. In this task, the authors propose a dataset of 775 handwritten flowchart images in Spanish and English languages. However, this dataset is unsuited for Flow2Code as it only consisted of hand-drawn flowchart images, with many samples consisting of only box drawings with no text, and the corresponding C codes were publicly unavailable. In this work, we consider Flow2Code as a sequence-to-sequence generation problem and address it using a state-of-the-art transformer-based technique in a data-driven manner. Further, we curate a dataset of 11.8K samples containing both digitized and handwritten flowchart images along with their corresponding Python codes to provide a more suitable benchmark for this task.

2.2 Large-scale pre-trained Language Models

The introduction of the transformer [32] architecture has brought a remarkable revolution in natural language processing. Further, to deal with the scarcity of labeled data and build a general-purpose model for a wide range of NLP applications, Radford et al. [25] proposed GPT, which is based on a transformer-decoder and pre-trained with an unlabeled pool of data in a self-supervised fashion. However, it follows a unidirectional autoregressive approach and is not suitable for tasks utilizing information from the entire sequence. Kenton et al. introduced BERT [17], a transformer-encoder-based method trained in a similar self-supervised fashion. BERT [17] follows a bidirectional autoencoder nature and is unsuitable for generation tasks that utilize information from the previously generated tokens in the sequence. To deal with the shortcomings of GPT [25] and BERT [17], Lewis et al. introduced BART [19], a denoising autoencoder that uses a bidirectional encoder and an auto-regressive decoder. These large-scale language models are often fine-tuned with a small set of labeled data for the supervised downstream task. In general, there are other well-explored pre-trained transformer-based methods such as T5 [26], MASS [29], ELECTRA [11], and RoBERTa [21]. In this work, we utilize CodeT5 [33], which adopts the encoder-decoder-based transformer model viz. T5 [26], and is pre-trained on programming language data.

2.3 Language Modeling for Code Generation

A significant amount of effort has been invested in automating software engineering using deep learning. Recent work has focused on transferable representations rather than task-specific ones. Pre-trained NLP models like BERT [17], GPT [25], and BART [19] have demonstrated transferability to programming languages, yielding positive results for a range of code-related tasks.

Feng et al. [14] introduced CodeBERT, which utilized BERT architecture pre-trained on programming language and natural language used in the software development domain, with masked language modeling objective. Guo et al. [15] proposed GraphCodeBERT as an improvement upon CodeBert by leveraging

Table 1. Statistics of the FloCo dataset.

Property	**Value**
Total number of samples	11,884
Avg. length of the program (in tokens)	46
Avg. length of the program (in lines)	4.6
Train set size	10,102
Test set size	1,188
Validation set size	594

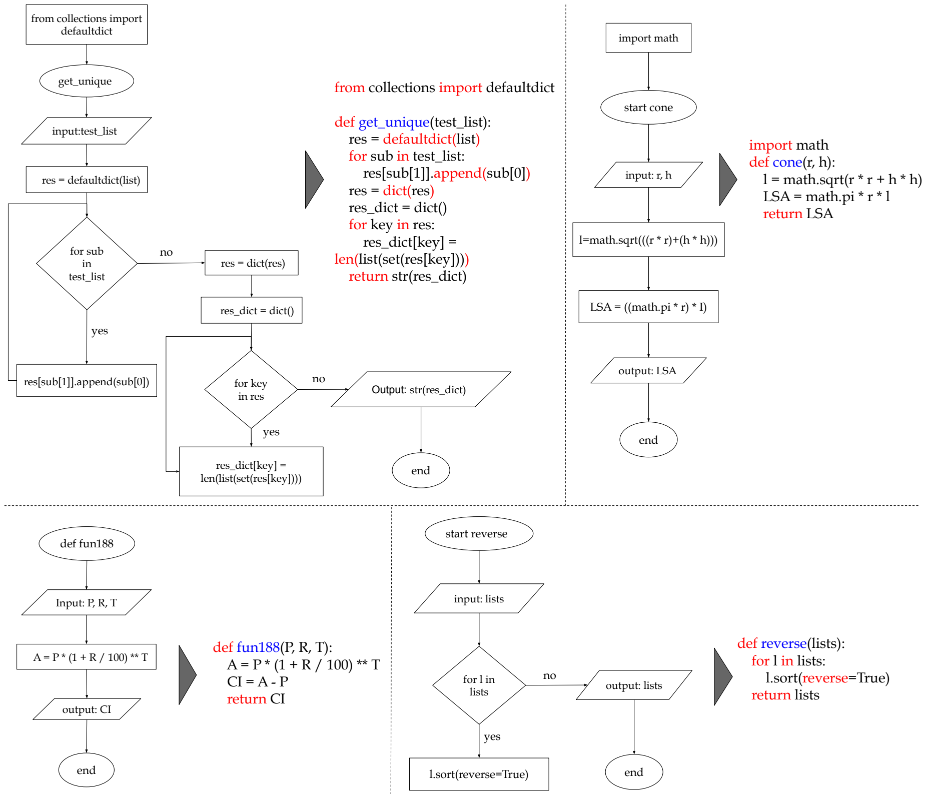
dataflow in source code through two additional pre-training tasks, predicting code structure edges, and aligning representations between source code and code structure. Ahmad et al. [2] introduced PLBART, a bidirectional and autoregressive transformer pre-trained on unlabeled natural language and programming language data, with denoising autoencoding objective, where the noising strategies employed were token masking, token deletion, and text infilling. Wang et al. [33] proposed CodeT5 by extending T5 [33] to programming languages. Similar to PLBART [2], it is a unified encoder-decoder transformer model, but it has task-specific fine-grain pre-training objectives such as masked span prediction, identifier tagging, masked identifier prediction, and bimodal dual generation objectives. As CodeT5 [33] has the advantage of task-specific pre-training strategies, we adopted it for our main method. We generated sequential encodings from flowchart images to treat Flow2Code as a sequence-to-sequence problem. We pre-trained the CodeT5 model with masked token modeling objective on a large number of logic-preserved augmented codes. Finally, we fine-tuned the pre-trained model for code generation.

3 FloCo: A novel dataset for Flowchart image to python Code conversion

We introduce a novel large-scale dataset for Flowchart images to python Code conversion. We refer to this dataset as FloCo. It contains 11,884 flowchart images and corresponding python codes. A selection of representative examples from the FloCo dataset is depicted in Figure 2. We make FloCo publicly available for download [2].

Flowchart-related research has been under-explored in the literature. However, there exist some related datasets such as (a) OHFCD dataset [5] has 419 handwritten flowcharts; however, it does not contain the corresponding codes as their focus is reading handwritten flowchart images and not code generation, (b) a more recent dataset namely FlowchartQA [31] introduces a synthetic dataset for question answering and reasoning on flowcharts. (c) in [23], authors introduced a collection of 775 handwritten flowchart images and corresponding C programming languages. However, codes for this dataset are not publicly avail-

[2] `https://vl2g.github.io/projects/floco/`



```
from collections import defaultdict

def get_unique(test_list):
    res = defaultdict(list)
    for sub in test_list:
        res[sub[1]].append(sub[0])
    res = dict(res)
    res_dict = dict()
    for key in res:
        res_dict[key] =
len(list(set(res[key])))
    return str(res_dict)
```

```
import math
def cone(r, h):
    l = math.sqrt(r * r + h * h)
    LSA = math.pi * r * l
    return LSA
```

```
def fun188(P, R, T):
    A = P * (1 + R / 100) ** T
    CI = A - P
    return CI
```

```
def reverse(lists):
    for l in lists:
        l.sort(reverse=True)
    return lists
```

Fig. 2. Samples from the proposed FloCo dataset. Each flowchart image is associated with the corresponding ground truth code.

able. Our new dataset, viz. FloCo has been introduced in this work to fill the research gap in the literature.

The FloCo dataset contains 11,884 flowchart images and python code pairs. The dataset has been generated by writing a few codes from scratch and gathering and cleaning codes from the MBPP (Mostly Basic Python Programs) [4], and code-to-text dataset of CodeXGleu [22] datasets. The digitized flowchart images corresponding to the codes are generated using the pyflowchart[3] and diagrams[4] libraries. FloCo is divided into train, test, and validation sets following an 85:10:5 ratio split respectively. Our data comprises a diverse collection of Python programs spanning a spectrum of complexity and uniqueness in their designated tasks. A few examples of these designated tasks include *calculating the N^{th} Fibonacci number, determining binomial coefficients, checking if a binary tree is balanced*, and *finding n^{th} Catalan number*. Detailed tatistics related to FloCo are provided in Table 1.

[3] https://pypi.org/project/pyflowchart/

[4] https://diagrams.mingrammer.com/

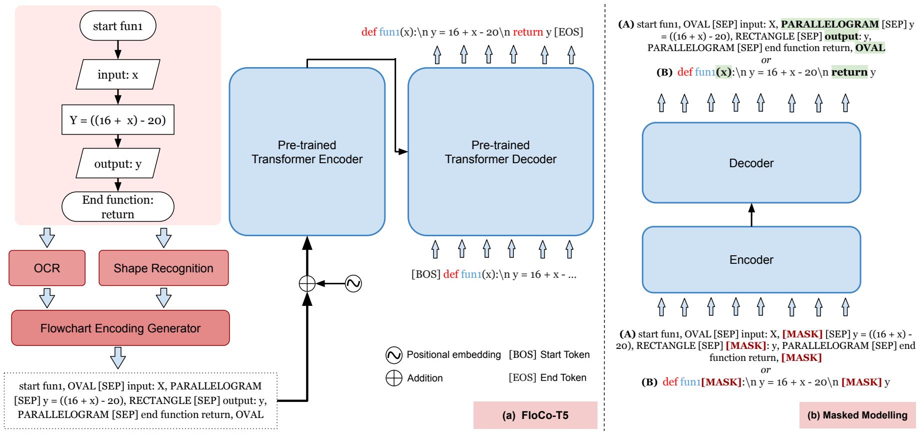

start fun1
input: x
Y = ((16 + x) - 20)
output: y
End function: return
OCR
Shape Recognition
Flowchart Encoding Generator
start fun1, OVAL [SEP] input: X, PARALLELOGRAM [SEP] y = ((16 + x) - 20), RECTANGLE [SEP] output: y, PARALLELOGRAM [SEP] end function return, OVAL
Pre-trained Transformer Encoder
Pre-trained Transformer Decoder
def fun1(x):\n y = 16 + x - 20\n return y [EOS]
[BOS] def fun1(x):\n y = 16 + x - ...
Positional embedding
Addition
[BOS] Start Token
[EOS] End Token
(a) FloCo-T5
(A) start fun1, OVAL [SEP] input: X, PARALLELOGRAM [SEP] y = ((16 + x) - 20), RECTANGLE [SEP] output: y, PARALLELOGRAM [SEP] end function return, OVAL
or
(B) def fun1(x):\n y = 16 + x - 20\n return y
Decoder
Encoder
(A) start fun1, OVAL [SEP] input: X, [MASK] [SEP] y = ((16 + x) - 20), RECTANGLE [SEP] [MASK]: y, PARALLELOGRAM [SEP] end function return, [MASK]
or
(B) def fun1[MASK]:\n y = 16 + x - 20\n [MASK] y
(b) Masked Modelling

Fig. 3. Overview of the proposed method, viz. FloCo-T5: (a). Given flowchart image is converted into sequence encoding using the off-the-shelf OCR techniques. The encoder of the pre-trained CodeT5 model takes flowchart encoding added with positional encodings as input. The Decoder initially takes the start token as input and has access to encoder output, then autoregressively generates the expected code word by word. **(b).** Shows the token mask modeling. Before fine-tuning, CodeT5 is pre-trained on a token mask modeling task, where some tokens of flowchart encoding are masked and reconstructed by the decoder in an unsupervised learning fashion [**Best viewed in color**].

4 Proposed Approach

The goal of Flow2Code is to generate code from a given flowchart image. We approach this task as a sequence-to-sequence generation problem involving two different modalities: image (flowchart) and text (code) and propose a framework, namely FloCo-T5 (Flowchart-to-Code T5 Model) that involves: (i) reading and converting the flowchart image into a sequence encoding, and then (ii) autoregressively generating code using the flowchart encoding. Figure 3 illustrates the proposed framework. We describe the two steps in the following subsections:

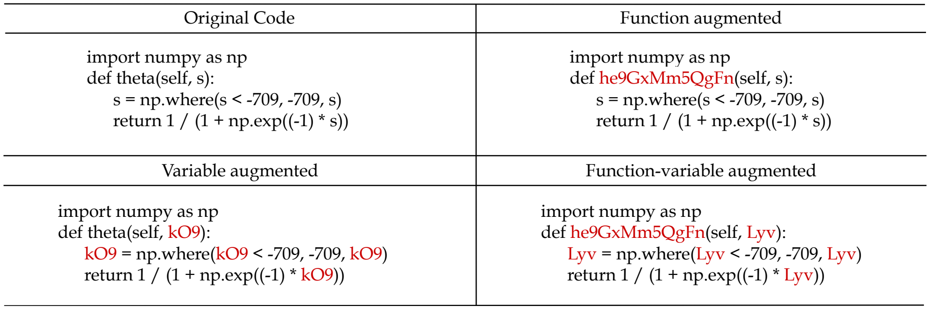
4.1 Flowchart Encoding Generation

In this step, we encode flowchart images into intermediate sequence encodings in the form of text. Given the flowchart image, we first detect and recognize the flowchart blocks, namely process (rectangle), decision (diamond), input/output (rhomboid), and terminal (oval), using the Hough transform-based shape detectors [7]. We further employ an off-the-shelf OCR method viz. easyOCR [1] to recognize the text within the boxes and on arrowheads for digitized flowchart images. We then match the recognized shapes and text using their respective coordinates, i.e., a text is paired with the name of a block only if the text coordinates lie within the shape coordinates. The final flowchart encoding is a

Table 2. Examples of different flowchart encoding. Refer to the main text for more details. A comparative study of different encodings is provided in Table 4.

Tuple encodings	String encodings	Modified string encodings
[('start fun1', 'OVAL'), ('input: X', 'PARALLELOGRAM'), ('y = ((16 + x) - 20)', 'RECTANGLE'), ('output: y', 'PARALLELOGRAM'), ('end function return', 'OVAL')]	{startfun1,OVAL},{input: X,PARALLELOGRAM},{y = ((16 + x) - 20),RECTANGLE},{output: y,PARALLELOGRAM},{end function return,OVAL}	start fun1, OVAL [SEP] input: X, PARALLELOGRAM [SEP] y = ((16 + x) - 20), RECTANGLE [SEP] output: y, PARALLELOGRAM [SEP] end function return, OVAL

Fig. 4. Data augmentation: Example of data augmentation used in code samples during pre-training of FloCo-T5. We propose three logic-preserving augmentations that include changing the function names, variable names, and both. Augmented names are highlighted in red color.



Original Code

```
import numpy as np
def theta(self, s):
    s = np.where(s < -709, -709, s)
    return 1 / (1 + np.exp((-1) * s))
```

Function augmented

```
import numpy as np
def he9GxMm5QgFn(self, s):
    s = np.where(s < -709, -709, s)
    return 1 / (1 + np.exp((-1) * s))
```

Variable augmented

```
import numpy as np
def theta(self, kO9):
    kO9 = np.where(kO9 < -709, -709, kO9)
    return 1 / (1 + np.exp((-1) * kO9))
```

Function-variable augmented

```
import numpy as np
def he9GxMm5QgFn(self, Lyv):
    Lyv = np.where(Lyv < -709, -709, Lyv)
    return 1 / (1 + np.exp((-1) * Lyv))
```

text sequence combining all the recognized text and shapes in the form of a key-value pair in the order in which they appear (from start to end). To this end, we experiment with three different strategies for encoding representation: (i) *tuple encodings*, wherein each step of the flowchart is represented as a tuple of the text and the box shape, each within quotes; (ii) *string encodings*, which eliminates quotes from text and shapes, and make use of braces to separate each step of the flowchart; and the optimized (iii) modified string encodings, where we utilize the [SEP] special tokens in the vocabulary of transformers to get rid of any additional braces or quotes and delineate each step of the flowchart. We provide an example for each of these encoding representations in Table 2. We experiment with all three encoding forms and compare their effectiveness on our target task in Table 4.

4.2 Code Generation

Inspired by the recent success of large-scale pre-trained code generation models, we adapt Code-T5 [33] – a transformer-based baseline trained for code generation, to our task. To this end, we initially pre-train it on a large number of logic-preserving augmented codes on the masked modeling objective in a self-supervised setting. The pre-training process adds knowledge of flowchart structure and code semantics to the model. Finally, we fine-tuned the pre-trained

Code-T5 on the training set of FloCo. The data augmentation, pre-training, and fine-tuning process are performed as follows:

Data Augmentation: In order to increase the size of the dataset while keeping the logic of codes intact, we explored data augmentation. This has been achieved by changing function names and variable names. We augmented the training subset of the FloCo dataset. Replacing all functions and variables with a specific set of characters would make the dataset biased. Therefore, the function and variable names were constructed randomly using uppercase/lowercase letters, underscore, and/or digits while keeping the naming conventions for the Python programming language in mind. The length of the function names was chosen randomly from the range of $4-13$; for variable names, the range was $1-3$. Thus, each program was augmented in three different ways: changing the function or variable names or changing both function and variable names together. Figure 4 depicts all the augmentations corresponding to a sample code. After augmentation, the train dataset size increased from $10,102$ to $40,408$. These $30,306$ augmented codes have been utilized at the pre-training stage of our method.

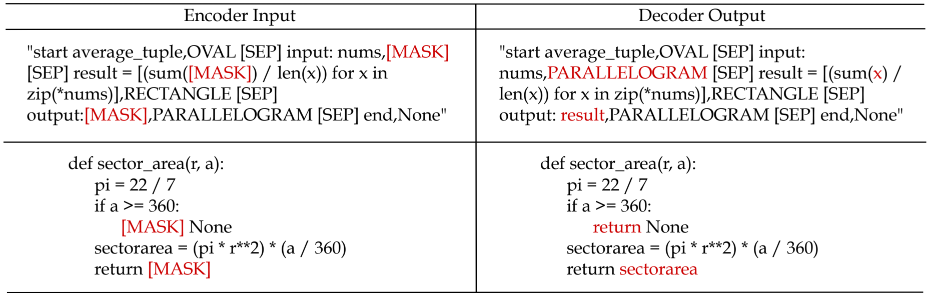
Masked Modeling Objective: Inspired by the success of the Masked Language Modeling (MLM) pre-training objective in BERT [17], we propose an analogous objective specific to our problem. We adopted the pre-trained CodeT5 model and trained it on the augmented codes and flowchart encodings of the train set of FloCo. Tokens in the pre-training dataset are masked randomly at a probability of 0.15, and we aim to optimize the loss associated with the reconstruction of the original sample, as shown below:

$$L_{mml}(E, \bar{E}) = -\sum_{t=1}^{N} log(e_t | e_{0:t-1}, \bar{E}). \tag{1}$$

where $E =< e_1, e_2 \ldots, e_N >$ and $\bar{E} =< f_r(e_1), f_r(e2), \ldots, f_r(e_N)) >$ represent the ground truth and masked encodings/code, respectively. $\bar{E}$ is obtained by applying the function $f_r(e_i)$ to the ground truth encoding, which randomly replaces token e_i with the mask token $[MASK]$ with a probability of 0.15. N, e_0 denotes the length of the flowchart encoding and start token, respectively. Figure 5 shows examples of masked modeling implemented for encoding and a code sample. For the encoding input, if we mask the shape of a block (*PARALLELOGRAM* in the given example), the model must be able to infer the correct shape based on the context and the pattern it has learned during training.

Fine-tuning: After pre-training FloCo-T5 on augmented data, we further fine-tuned it on the training data of FloCo, for Flow2Code task. Figure 3 (a) shows the training pipeline; the given flowchart image is first converted into sequence encoding by detecting shapes and the text inside the shapes using an off-the-shelf OCR technique. Positional encodings are added to the flowchart encodings before feeding them to the encoder. The decoder has access to the output of the encoder. It starts with a start token, and auto-regressively generates

Fig. 5. Masked Modelling: Example encoder inputs and decoder outputs during mask token prediction of FloCo-T5.



Encoder Input	Decoder Output
"start average_tuple,OVAL [SEP] input: nums,[MASK] [SEP] result = [(sum([MASK]) / len(x)) for x in zip(*nums)],RECTANGLE [SEP] output:[MASK],PARALLELOGRAM [SEP] end,None"	"start average_tuple,OVAL [SEP] input: nums,PARALLELOGRAM [SEP] result = [(sum(x) / len(x)) for x in zip(*nums)],RECTANGLE [SEP] output: result,PARALLELOGRAM [SEP] end,None"

def sector_area(r, a):
pi = 22 / 7
if a >= 360:
[MASK] None
sectorarea = (pi * r**2) * (a / 360)
return [MASK]
	def sector_area(r, a):
	pi = 22 / 7
	if a >= 360:
	return None
	sectorarea = (pi * r**2) * (a / 360)
	return sectorarea

code token-by-token. To this end, during fine-tuning, we employed a language modeling loss expressed as follows:

$$L(X, E) = -\sum_{t=1}^{M} log(x_t | x_{0:t-1}, E). \tag{2}$$

where $X = < x_1, x_2, \ldots, x_M >$ denotes the ground truth code. Additionally, M, and x_0 represent the length of the code and start token, respectively. Note that during both the pre-training and the fine-tuning stages, we include different flowchart box shapes as a special token in the transformer model's vocabulary.

5 Experiments and Results

In this section, we present an extensive experimental analysis on the FloCo benchmark to verify the efficacy of our proposed model.

5.1 Evaluation metrics

Following the code generation literature [2], we evaluated the performance of our baselines and proposed model using the following three metrics:

i. **BLEU [24]**: a widely used word-overlap metric for assessing the quality of machine-translated text by comparing the n-grams of the generated code to the reference (ground truth) code and counting the number of matches.
ii. **CodeBLEU [27]**: a specialized metric that evaluates the quality of generated code, taking into account syntactical and logical correctness and the code's structure as reflected in the abstract syntax tree and data flow, in addition to comparing n-grams.
iii. **Exact Match (EM)**: a binary metric that checks if the generated code sequence is exactly the same as the ground-truth code.

Table 3. On the FloCo test set, we compared FloCo-T5 to competitive transformer-based baselines and found that our method achieved higher scores for all metrics.

Method	BLEU	CodeBLEU	EM
Vanilla Transformer [32]	10.3	26.8	0.0
BART [19]	31.1	40.7	2.2
PLBART [2]	55.7	63.7	19.4
CodeT5 [33]	63.8	71.8	17.8
FloCo-T5	**67.4**	**75.7**	**20.0**

5.2 Baseline Models

To evaluate the effectiveness of the proposed method, we compared it against the following four competitive baselines:

Vanilla Transformer [32] is the attention-based encoder-decoder architecture upon which the transformer-based pre-trained models are built. By comparing the proposed method with this baseline, we can observe the specific advantages of pre-training.

BART [19] is a pre-trained, bidirectional, autoregressive encoder-decoder architecture that was pre-trained on unlabelled natural language data and optimized using reconstruction loss. The noising techniques used were token masking, token deletion, text infilling, sentence permutation, and document rotation.

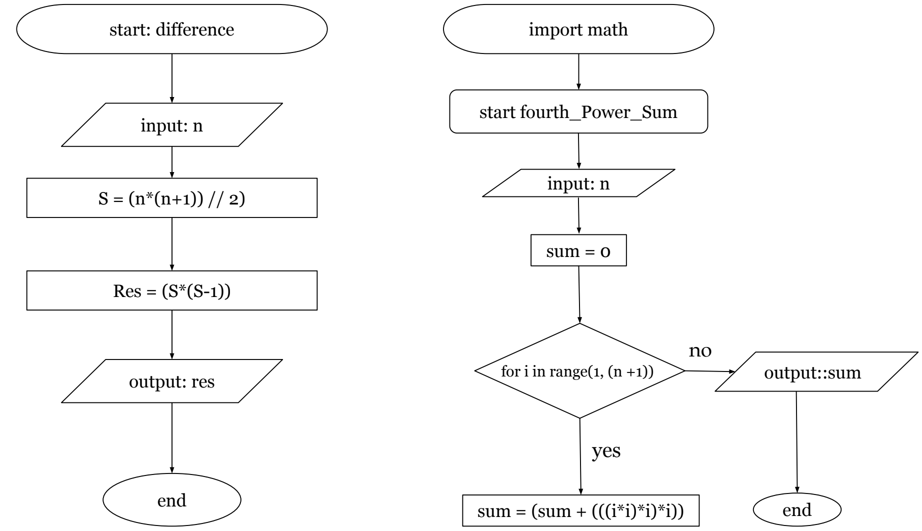
PLBART [2] is an extension of BART and was pre-trained on a large-scale dataset containing unlabelled natural language and programming language data. The pre-training objective was denoising autoencoding, and the noising strategies used were token masking, deletion, and infilling.

CodeT5 [33] adopted the T5 (pre-trained on natural language) architecture and was pre-trained on natural language and programming language data. The pre-training objectives were span prediction, identifier tagging, masked identifier prediction, and bimodal dual generation.

By comparing our proposed method with these baselines, we can observe how our method outperforms them and understand how it leverages the pre-training.

5.3 Implementation details for Reproducibility

FloCo-T5 is implemented using the Huggingface library [34] and utilizes the implementation of CodeT5 [33], using the 'Salesforce/codet5-base' pre-trained model available on Huggingface. The model contains 222.9 million trainable parameters. It consists of 12 encoder, 12 decoder layers, and 12 attention heads in each layer. The input encodings are truncated or padded to a maximum length of 512 tokens. We optimize training using the Adam [18] optimizer with a learning rate of $1e-5$, a warmup for 2450 steps, and a batch size of 16. We use the same training configuration in both the pre-training and fine-tuning stages. All the baselines were trained on a single NVIDIA A6000 GPU with 48 GB VRAM.



Ground Truth

```
def difference(n):

    S = (n * (n + 1)) // 2
    res = S * (S - 1)
    return res
```

```
def fourth_Power_Sum(n):
    sum = 0
    for i in range(1, n + 1):
        sum = sum + (i * i * i * i)
    return sum
```

PLBART

```
res = n * (n + 1)) // 2
res = s * (S - 1)
return res
```

```
def math(n):
    sum = 0 for i in range(1, n + 1):
    sum = sum + i * i * i
    return sum
```

CodeT5

```
def difference(n):

    s = n * n + 1
    res = s * (s - 1)
    return res
```

```
def fourth_power_sum(n):

    sum = 0
    for i in range(1, n+1):
        sum = sum + (i * i) * i
    return sum
```

FloCo-T5 (Ours)

```
def difference(n):

    S = (n * (n + 1)) // 2
    res = S * (S - 1)
    return res
```

```
def _fourthPowerSum(n):

    sum = 0
    for i in range(1, n+1):
        sum = sum + (i * i * i * i)
    return sum
```

Fig. 6. Qualitative comparison: Results on two flowchart images using PLBART, CodeT5 and our method. Errors are highlighted in dark red color. The codes generated by our method are similar to the ground truth as compared to the PLBART and CodeT5. CodeT5 has fewer errors as compared to PLBART.

Table 4. Comparision of different flowchart encoding representations on the performance of FloCo-T5.

Method	BLEU	CodeBLEU	EM
Tuple encodings	16.7	37.7	0.2
String encodings	50.1	63.4	11.1
Modified string encodings (Ours)	**67.4**	**75.7**	**20.0**

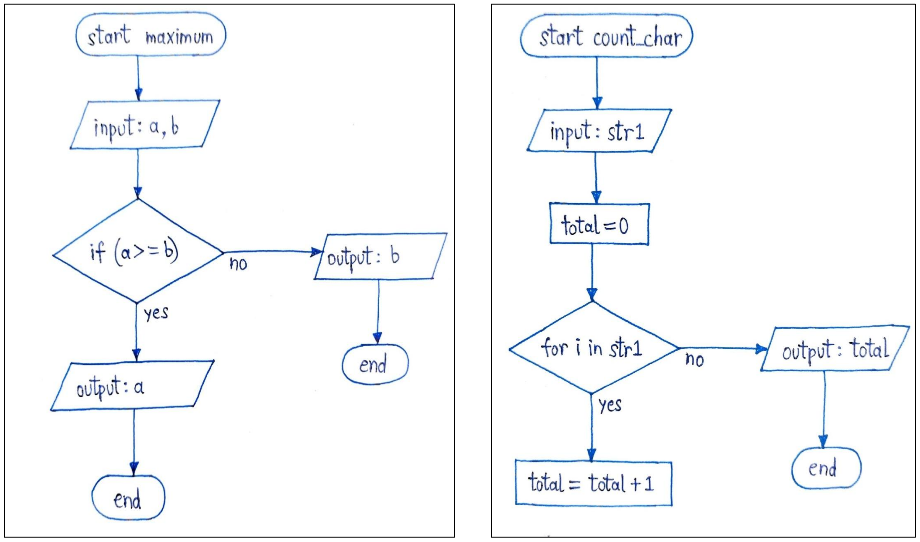
The training process requires nearly 12 hours to reach convergence. We make our implementation available here: `https://vl2g.github.io/projects/floco/`.

5.4 Results and discussions

We evaluate our method on the proposed FloCo dataset and compare it against competitive baselines, namely vanilla transformer [32], BART [19], PLBART [2], and CodeT5 [33]. Table 3 shows the performance of the implemented baselines and proposed FloCo-T5 on three evaluation metrics. Vanilla Transformer [32] is trained from scratch in contrast to other baselines, pre-trained on large-scale unlabelled data with different self-supervised pre-training objectives. Hence, Vanilla Transformer lacks the understanding of language and programming semantics and structure, resulting in the lowest performance for all the metrics. BART [19] is pre-trained on natural language text and thus, has a better understanding of the semantics and structure of the sequential data, as natural text also has rules, structure, and other syntactical properties. It results in better performance as compared to the Vanilla Transformer for all of the metrics. PLBART is pre-trained on the natural text and programming language, which means it has a better understanding of code structure and semantics, resulting in better performance compared to BART and Vanilla Transformer on all metrics. CodeT5 [33] is pre-trained with programming-language-specific, fine-grained identifier-aware denoising tasks, which help in exploiting code semantics and structure in a more exquisite way, resulting in significant improvement over other baselines. In the proposed FloCo-T5, we adopted a pre-trained CodeT5 model, which has task-specific knowledge, and further pre-trained it on augmented training samples for the mask token generation task. As expected, FloCo-T5 outperforms all baselines for all the metrics used for evaluation, showing the efficacy of the proposed code augmentation and pre-training strategy.

Figure 6 shows the generated codes for two flowchart samples. We compare the ground truth codes with the ones generated from PLBART, CodeT5, and our method. FloCo-T5 is able to generate codes syntactically correct codes, which are similar to the ground truth codes, while other baselines fall short in generating correct codes. This observation is same across other test samples as numerically summarized by Table 3.

We further conducted an experiment with three flowchart image encoding methods, shown in Table 2, and results presented in Table 4. The modified string encoding method utilized a [SEP] token to separate each step of the flowchart,



Ground Truth

```
def maximum(a, b):
    if a >= b:
        return a
    else:
        return b
```

```
def count_char(str1):
    total = 0
    for i in str1:
        total = total + 1
    return total
```

FloCo-T5 (Ours)

```
def maximum(a, b):
    if a < b:
        return b
    return a
```

```
def count_char(str1):
    total = 0
    for i in str1:
        total += total * 4
    return total
```

Fig. 7. Results on hand-drawn flowchart images using FloCo-T5. Errors are highlighted in dark red color. We observe an error in the program due to incorrect recognition of the handwritten character '*' by our OCR module.

and removed extra braces, enhancing the preservation of the flowchart's structure, and consequently outperforming other encoding methods.

Can the proposed approach work for hand-drawn Flowchart Images? We evaluated FloCo-T5 on hand-drawn flowchart images using 40 samples created by three human annotators. Flowchart block detection and recognition were performed with OpenCV [7]. For handwritten text recognition, we employed CRAFT text detection [6] and TrOCR text recognition [20]. FloCo-T5 achieved a BLEU score of 21.4% and a CodeBLEU score of 34.6% on these hand-drawn flowcharts. Fig. 7 displays Python codes generated for two sample hand-drawn flowcharts. These results indicate our approach's suitability for hand-drawn flowcharts, and performance can be significantly enhanced with advances in handwritten text recognition.

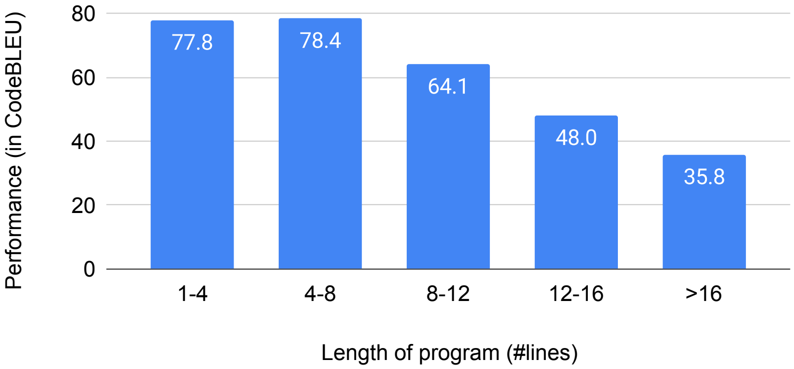


Fig. 8. Performance of FloCo-T5 across various program lengths.

Limitations: We observed that code generation performance of our model is higher for shorter programs (<≈ 12 lines) but drops for longer programs (>≈ 15 lines) due to the dataset's bias towards shorter programs (average length: 4.6 lines) as shown in Figure 8. To address this issue, we propose increasing the number of training samples for longer programs. Future work will focus on expanding the FloCo dataset to include longer and more complex code and researching information flow in state and block diagrams.

6 Conclusion

We introduced the FloCo-T5 framework for generating Python code from flowchart images and presented the FloCo dataset to benchmark the Flow2Code task. Flow2Code is modeled as a sequence-to-sequence problem, where flowcharts are first encoded by detecting the shapes of blocks and reading the text within, and further transformed into code using competitive transformer baselines. FloCo-T5's task-specific pre-training results in significant improvements over related baselines. The recent advancements in Large Language Models (LLMs), such as ChatGPT, have revolutionized the field of code generation, and they can be adapted to solve our task. However, ensuring that these massive models have not seen our test data is not a trivial task. Furthermore, despite these advancements, we firmly believe that our dataset can be used to study open problems such as development of lightweight and interpretable models for generating code from flowchart images. We leave these as future directions to work on.

Acknowledgements: This work was partly supported by MeitY, Govt. of India (project number: S/MeitY/AM/20210114). Yogesh Kumar is supported by a UGC fellowship.

References

1. Easy ocr (2022), available at: https://pypi.org/project/easyocr/1.6.2/
2. Ahmad, W., Chakraborty, S., Ray, B., Chang, K.W.: Unified pre-training for program understanding and generation. In: Proc. NAACL-HLT (2021)
3. Akermi, I., Heinecke, J., Herledan, F.: Transformer based natural language generation for question-answering. In: Proceedings of the 13th International Conference on Natural Language Generation (2020)
4. Austin, J., Odena, A., Nye, M.I., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C.J., Terry, M., Le, Q.V., Sutton, C.: Program synthesis with large language models. CoRR **abs/2108.07732** (2021)
5. Awal, A.M., Feng, G., Mouchère, H., Viard-Gaudin, C.: Handwritten flowchart dataset (ohfcd). Document Recognition and Retrieval XVIII, Jan 2011, San Fransisco, United States. pp.7874 - 78740A, 2011, ⟨10.1117/12.876624⟩ (2011)
6. Baek, Y., Lee, B., Han, D., Yun, S., Lee, H.: Character region awareness for text detection. In: Proc. CVPR (2019)
7. Bradski, G.: The OpenCV Library. Dr. Dobb's Journal of Software Tools (2000)
8. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language models are few-shot learners. In: Proc. NeurIPS (2020)
9. Carton, C., Lemaitre, A., Coüasnon, B.: Fusion of statistical and structural information for flowchart recognition. In: Proc. ICDAR (2013)
10. Chaudhuri, A.: Flowchart and algorithm basics: The art of programming. Mercury Learning and Information (2020)
11. Clark, K., Luong, M.T., Le, Q.V., Manning, C.D.: ELECTRA: Pre-training text encoders as discriminators rather than generators. In: Proc. ICLR (2020)
12. Cook, D.: Flowgorithm (2022), available at: http://www.flowgorithm.org/
13. Fang, J., Feng, Z., Cai, B.: Drawnnet: Offline hand-drawn diagram recognition based on keypoint prediction of aggregating geometric characteristics. Entropy **24**(3) (2022)
14. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M.: CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the ACL: EMNLP 2020 (2020)
15. Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Shujie, L., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al.: Graphcodebert: Pre-training code representations with data flow. In: Proc. ICLR (2020)
16. Herrera-Camara, J.I., Hammond, T.: Flow2code: from hand-drawn flowcharts to code execution. In: Proceedings of the Symposium on Sketch-Based Interfaces and Modeling (2017)
17. Kenton, J.D.M.W.C., Toutanova, L.K.: Bert: Pre-training of deep bidirectional transformers for language understanding. In: Proc. NAACL-HLT (2019)
18. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: Bengio, Y., LeCun, Y. (eds.) Proc. ICLR (2015)
19. Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., Zettlemoyer, L.: Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: Proc. ACL (2020)

20. Li, M., Lv, T., Chen, J., Cui, L., Lu, Y., Florencio, D., Zhang, C., Li, Z., Wei, F.: Trocr: Transformer-based optical character recognition with pre-trained models. arXiv preprint arXiv:2109.10282 (2021)
21. Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V.: Roberta: A robustly optimized BERT pretraining approach. CoRR **abs/1907.11692** (2019)
22. Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong (YIMING), M., Zhou, M., Duan, N., Sundaresan, N., Deng, S.K., Fu, S., Liu, S.: Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv (2021)
23. Montellano, C.D.B., Garcia, C.O.F.C., Leija, R.O.C.: Recognition of handwritten flowcharts using convolutional neural networks. International Journal of Computer Applications (2022)
24. Papineni, K., Roukos, S., Ward, T., Zhu, W.J.: Bleu: a method for automatic evaluation of machine translation. In: Proc. ACL (2002)
25. Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al.: Improving language understanding by generative pre-training. OpenAI (2018)
26. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J., et al.: Exploring the limits of transfer learning with a unified text-to-text transformer. J. Mach. Learn. Res. (2020)
27. Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., Ma, S.: Codebleu: a method for automatic evaluation of code synthesis. CoRR **abs/2009.10297** (2020)
28. Schäfer, B., Stuckenschmidt, H.: Arrow r-cnn for flowchart recognition. In: Proc. ICDAR Workshop (2019)
29. Song, K., Tan, X., Qin, T., Lu, J., Liu, T.Y.: Mass: Masked sequence to sequence pre-training for language generation. In: Proc. ICML (2019)
30. Supaartagorn, C.: Web application for automatic code generator using a structured flowchart. In: 2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS) (2017)
31. Tannert, S., Feighelstein, M., Bogojeska, J., Shtok, J., Staar, A.A.P., Karlinsky, A.S.J.K.L.: Flowchartqa: The first large-scale benchmark for reasoning over flowcharts. Document Intelligence Workshop @ KDD (2022)
32. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Proc. NeurIPS (2017)
33. Wang, Y., Wang, W., Joty, S., Hoi, S.C.: Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proc. EMNLP (2021)
34. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T.L., Gugger, S., Drame, M., Lhoest, Q., Rush, A.M.: Transformers: State-of-the-art natural language processing. In: Pro. EMNLP: System Demonstrations (2020)
35. Wu, X.H., Qu, M.C., Liu, Z.Q., Li, J.Z., et al.: Research and application of code automatic generation algorithm based on structured flowchart. Journal of Software Engineering and Applications (2011)