

Noise-and Lighting-Robust Handwritten Code Recognition with Syntax-Aware Error Correction for Braille Output

Varshini.A¹, S.Gopinathan²

¹ Ph.D Research Scholar, Department of Computer Science, University of Madras, India.; e-mail : varshini2699@gmail.com

² Professor, Department of Computer Science, University of Madras, India; e-mail : gnathans2002@gmail.com

* Corresponding Author : S.Gopinathan 

HOW TO CITE:

Varshini.A, S.Gopinathan
(2026). Noise-and Lighting-Robust
Handwritten Code Recognition
with Syntax-Aware Error Correc-
tion for Braille Output.
International Journal of Special Edu-
cation, 41(19s), 588 - 610

COPYRIGHT STATEMENT:

Copyright: © 2026 Authors.
Open access publication under the
terms and conditions
of the Creative Commons Attribution
(CC BY)
license (<http://creativecommons.org/licenses/by/4.0/>).

ABSTRACT:

Visually impaired students are greatly challenged in accessing programming materials due to the large number of programming materials existing as handwritten images. The variations of lighting conditions, noises, skews, and handwriting styles make it difficult for conventional Optical Character Recognition (OCR) to identify and limit the possibilities of creating accurate Braille transcriptions. In this paper, we propose an approach of converting handwritten programming code into a Braille-accessible format. To achieve high-quality recognition accuracy, the proposed approach utilizes a combination of preprocessing of images, handwriting text recognition, and syntax validation. Furthermore, the system uses a token validation method and analyzes the structure of the source code by utilizing the abstract syntax trees to correct structural errors within the source code. After validation, the output is translated into technical Braille based on unified English Braille computing rules. Outputs include Unicode Braille and Braille-ready formats.

Keywords: Braille computing; assistive technology; syntax correction; handwriting recognition; handwritten code recognition.

Introduction

Programming education is now considered a significant part of contemporary computer science courses. Visually Impaired Learners (VILs), however, encounter numerous barriers when attempting to access handwritten programming notes/tutorial materials from

their peers or mobile devices. Many learning environments continue to use

Existing systems focus on accessible printed or digital code. Standard Optical Character Recognition (OCR) systems are not capable of processing handwritten programming syntax or images that may be low-quality due to degradation caused by capture under real-world conditions. In addition, mobile images are

subject to many different forms of degradation, including, but not limited to, uneven illumination, shadows, noise, blurring, and geometric distortion, which further degrade the ability of OCR systems to recognize content.

Minor errors in symbols in programming languages can result in syntactic failure when transcribing content into Braille for VILs. For those relying on Braille for understanding programming concepts, this creates a major barrier. In order to address the above-described issues, this paper presents a novel,

robust handwritten code recognition framework that combines image enhancement techniques with syntax-aware error correction. The proposed system provides improved recognition under variable illumination conditions while also maintaining syntactical correctness prior to Braille transcription.

This paper's major contributions include

- A Lighting-Robust Preprocessing Framework for Handwritten Programming Images.
- A Handwriting Recognition Pipeline Preserving Code Structure.
- A Syntax-Aware Error Correction Module Utilizing Lexical and Syntactic Validation.
- A Technical Braille Transcription Module Supporting Computing Braille Standards.
- A Dataset of Handwritten Programs in C, C++, Java, and Python Captured Under Real-World Conditions.

2. Related Work

Recent related work in Offline Handwritten Text Recognition (HTR) has been driven by architectures that integrate deep learning including Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Bidirectional LSTM networks and Connectionist Temporal Classification decoders [1]-[9]. Early systems used RNNs to recognize unconstrained sequences of handwritten text and greatly improved tasks of document analysis and pattern recognition [1]. Higher dimensional RNNs further improved recognition by exploiting spatial dependencies within handwritten documents [2]. CTC also enabled learning

sequences without requiring explicit segmentation at character level [3]. Hybrid CNN and RNN architectures became dominant subsequently for feature extraction and sequential decoding and greatly improved accuracy on diverse handwriting datasets [4]-[7]. Additional studies also strengthened contextual learning by incorporating n-grams and recurrent layers and thus improved performance under complex handwriting conditions [7], [8]. More recently encoder-decoder and sequence to sequence frameworks have shown better performance modelling dependencies that stretch over long distances within handwritten text. Further comprehensive surveys stress that benchmark datasets, augmentation strategies and robust evaluation protocols are important for improving generalization capability of modern HTR systems [10]-[12]. Despite these advances, current frameworks for HTR are mainly optimized for natural language documents and are not designed to recognize handwritten code. Programming languages have syntax sensitive structures, special symbols, indentation, brackets, operators and naming conventions that are fundamentally different from normal handwritten text. Systems that convert page images into sequences of text perform efficiently [13]; self-training strategies have also been proposed to use unlabelled handwriting data to improve performance [14]. However, approaches like these do not model syntax of programming languages nor preserve structural semantics necessary for accurate recognition of handwritten source code.

Research on Optical Character Recognition (OCR) for programming languages has mainly focused on extracting source code from printed documents, screenshots from Integrated Development Environments (IDEs) or digital images [15] and [16]. Techniques assume clean layouts, consistent fonts and high contrast digital images and so they are not suitable for code acquired by mobile devices that is handwritten and acquired under different lighting conditions. Consequently, accuracy drops sharply when there are handwriting variations, perspective distortion, shadows and background noise. This shows that robust preprocessing and techniques specific to handwriting are needed. Parallel research has also worked on accessibility for people

who are visually impaired. Some studies have looked into assistive technologies that support navigation, debugging and workflows for software development [17]-[21]. Other researchers have proposed accessible environments with special adaptations of IDEs, tactile interfaces, structured methodologies for programming and assistive tools that help blind programmers better understand structure of software better [22]-[27]. While these systems greatly improve accessibility during software development generally, they assume that digital source code is already available and do not address automatic recognition of handwritten programming documents.

Recent attention has also been paid to post recognition correction. Machine learning and neural language model OCR correction methods have shown improvements at correcting errors in text documents [28], [29]. Similarly, systems for translating text to Braille have become increasingly important for improving accessibility for people who are blind by converting digital text into standard Braille representations [30]. However, current frameworks for Braille translation process text that has already been correctly recognized and do not deal with unique challenges of handwritten code such as preserving syntax, programming symbols and technical notation.

Research Gap: Progress has been significant in handwritten text recognition using OCR and for accessibility to code but no current framework unifies all these components into a pipeline specifically for handwritten source code. Current HTR models are fine tuned for natural language rather than syntax; OCR techniques target printed code; accessibility tools assume that code is available digitally; correction methods for OCR are not syntax aware for programming languages; and existing translators of Braille do not preserve structural and semantic features of handwritten source code. There remains a large research gap in developing an end-to-end system that integrates preprocessing resistant to illumination changes, deep recognition of handwritten code, correction at multiple levels using lexical analysis and Abstract Syntax Tree (AST) validation along with rescoring using beam search and transcription in Unified English Braille (UEB)

for technical Braille. Proposed framework closes this gap by providing an integrated solution that accurately converts handwritten programming code into accessible representations of Braille; this supports education that is inclusive for visually impaired learners.

3. Contributions and Novelty

While several recent studies have improved handwriting recognition and accessible technology, there has been very little research on how to convert handwritten code into Braille representations. Many studies have shown that handwritten code images taken with a smartphone can be degraded due to lighting variations, noise, slant, and differences in handwriting style, all of which negatively impact recognition accuracy. Additionally, Optical Character Recognition (OCR) technology does not consider the syntax of programming or its structural layout.

This study addresses these issues through proposing a robust handheld code recognition and Braille transcription framework that combines image enhancement, structure-aware recognition, and syntax-aware correction. The main contributions of this study include:

1. Improved Quality of Handwritten Code Images

A single pipeline for improving the quality of handwritten code images captured in real-world settings is proposed. This pipeline contains the following processes: normalizing illumination, adaptive thresholding, CLAHE (Contrast Limited Adaptive Histogram Equalization) for enhancing contrast, and deskewing documents for correcting slanted handwriting.

These improvements enhance the visibility and legibility of handwritten characters before they are recognized.

2. A Structure-Aware Recognition System

Unlike traditional OCR systems that treat text as simply an ordered sequence of characters, the proposed recognition system retains the line and indentation structure of programming languages. The retention of structure is important for

accurately identifying the groups of statements that form a block of code and the control flow of a program.

3. A Multi-Level Error Correction System

A multi-level error correction method is proposed to handle the common types of errors found in handwritten code. This multi-level error correction method involves three levels of error correction: lexical validity, bracket consistency, and AST parsing for detecting and correcting syntactical errors.

4. Language-Specific Rescore Method

In addition to improving recognition accuracy, a beam search rescoring is employed to guide the rescoring process based upon a model of the target programming language. This allows for the evaluation of many possible candidate sequences of tokens and selection of the sequence with the highest probability of being syntactically correct given the target programming language's syntax.

The system design is shown graphically as follows:

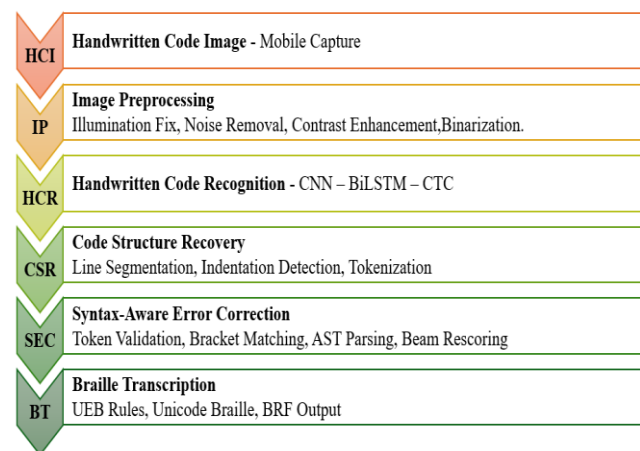


Figure 1. System Design Framework of Handwritten Code Recognition and Braille Transcription

recovery, (iv) syntax-aware error correction, and (v) Braille transcription. Each module was developed to enhance recognition accuracy and correct syntax before converting the input image into Braille.

4.2. System Workflow

The processing pipeline for the proposed system has been depicted in the figure below:

5. Technical Braille Translation Module

Once the source code is determined to be correct, it is then translated into standard computer Braille representations. The translation module supports the UEB (Unified English Braille) computing conventions and produces both Unicode Braille and Braille Ready Format (BRF) output for use with assistive devices like Braille displays and embossers.

4. Proposed Methodology

4.1. System Architecture

The proposed system framework will take the user's hand-written code images and convert those images into Braille for accessibility purposes. The proposed architecture includes several key elements: image processing; a handwriting recognition module; a syntax-aware correction module; and a Braille transcription module.

The system design framework of handwritten code recognition and Braille transcription is shown in Figure 1. The system will capture images of the user's hand-written programming code using a mobile device and then generate an accessible Braille representation from that image. The system has five main components: (i) image processing, (ii) handwritten code recognition, (iii) code structure

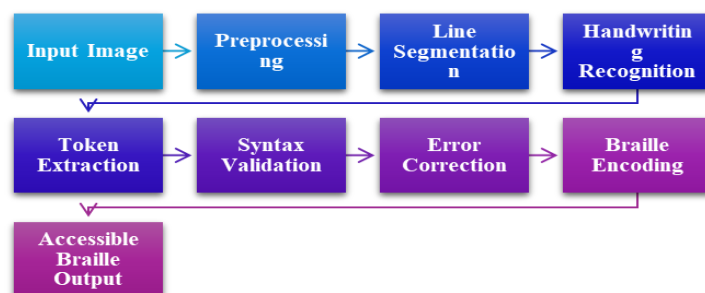


Figure 2. Handwritten Code Recognition and Braille Generation System's Processing Pipeline.

original programming errors. Finally, the corrected source code will be translated into technical Braille using the United States Unified English Braille Computing Rules to produce both Unicode Braille and Braille Ready Format output formats.

4.3. Image Preprocessing

In order for handwritten code images taken with a mobile device to be usable, they need to undergo preprocessing. Most images of handwritten code will have noise, shadow or some type of light variation that affects how well the recognition performs on these images; therefore, multiple forms of preprocessing are used.

The proposed system's processing pipeline is shown in Fig. 2. This begins with preprocessing of handwritten images of code to eliminate illumination variation and remove noise. These processed images are then passed through a handwriting recognition module, which uses a deep neural network model to extract sequential textual representations from the input images. Once the code has been identified, it will be parsed to identify structural relationships (i.e., indentation levels and token boundaries). Once the structural relationships have been identified, they will be used by a syntax-aware correction module to validate the recognized tokens and to make any necessary corrections to the

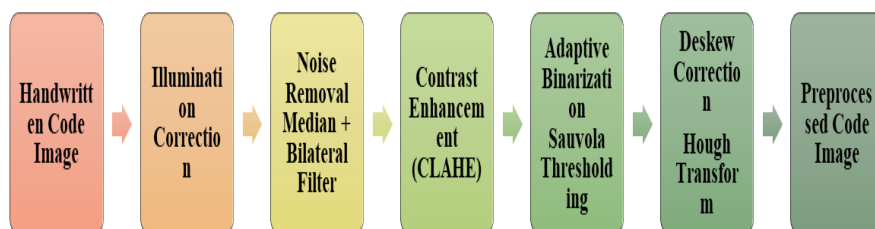


Figure 3. Steps involved in the pre-processing of handwritten code images for enhancement prior to recognition.

transform to align the document for accurate recognition.

1. Illumination Normalization

A technique called Retinex-based illumination normalization can be used to remove the variation in lighting from an image. The normalized image, I_{enh} , can then be expressed as

$$I_{\{enh\}} = \log(I) - \log(G * I)$$

(1)

where,

I is the original image, and

Steps involved in the pre-processing of handwritten code images are shown in Fig. 3. The process involves illumination normalization through Retinex or adaptive gamma correction to account for varying levels of light; removal of noise from the image by median and bilateral filtering to preserve the edges of the characters; application of contrast limited adaptive histogram equalization (CLAHE) to improve the legibility of the handwritten characters; conversion of the image to a binary representation using adaptive Sauvola thresholding; and finally, applying skew correction using the Hough

where,

$m(x,y)$ = local mean

$s(x,y)$ = local standard deviation

R = dynamic range

k = tuning constant.

5. Deskew Correction

The skew angle θ is calculated using the Hough transform and can then be corrected as

$$I_{\text{deskew}} = \text{Rotate}(I_{\text{bin}}, -\theta) \quad (4)$$

4.4. Handwritten Code Recognition

After preprocessing of the document is done, the document is separated into individual lines of code

G is a Gaussian smoothing filter.

2. Noise Removal

Impulsive noise can be removed using median filters that preserve edge features:

$$I_{\text{med}(x,y)} = \text{median}\{I(i,j)\} \quad (2)$$

where (i,j) are neighboring pixel values.

Further removal of noise and preservation of structural edges is possible using bilateral filters.

3. Contrast Enhancement

Improving visibility of characters in the image through redistribution of local intensity values can be achieved through Contrast Limited Adaptive Histogram Equalization (CLAHE).

4. Adaptive Binarization

Using Sauvola thresholding, gray scale images can be converted to binary format using a locally adaptive threshold:

$$T(x,y) = m(x,y) \left[1 + k \left(\left(\frac{s(x,y)}{R} \right) - 1 \right) \right] \quad (3)$$



Figure 4. CNN-BiLSTM-CTC architecture used for recognizing handwritten code.

A handwritten code recognition system was developed using a combination of a Convolutional Neural Network (CNN) and Bidirectional Long Short-Term Memory (BiLSTM) to recognize handwritten code.

1. CNN Feature Extraction

The CNN will extract spatial features from the input image:

$$F = \text{CNN}(I_{\text{deskew}}) \quad (5)$$

where,

F = The feature map that has been extracted.

This architecture in Fig. 4 is utilized as the deep learning model for handwritten code recognition. The first layer of this model is convolutional neural networks (CNNs), which are able to extract the spatial features of the input image. These feature maps are then transformed into sequential representations and fed into bidirectional long short-term memory (BiLSTM) layers to recognize the contextual relationships between characters. The final layer of this model is the Connectionist Temporal Classification (CTC) decoder, which can convert the predicted probability values into sequences of characters without the need for character segmentation.

4.5. Code Structure Reconstruction

Structural coding formatting (e.g., indentation and block-level nesting) is a primary aspect of most programming languages.

To rebuild the coding structure, the system follows these steps:

1. Line separation
2. Estimate Indentation
3. Tokenize
4. Identify Keywords

Indents are estimated using horizontal distances among consecutive lines to estimate indents. Tokens separate code elements (i.e., identifiers, operators, etc. and punctuation symbols).

4.6. A Syntax-Aware Mechanism for Correcting Recognition Errors in Programming

The proposed system will use a multi-phase, syntax-aware mechanism to correct recognition errors that may produce syntactically invalid programs.

2. Sequential Modeling

The BiLSTM layers will model the sequential nature of the code as well as its contextual dependencies:

$$H_t = BiLSTM(F_t) \quad (6)$$

where,

H_t represents the hidden representation of the input at each time-step t .

3. CTC Decoding

Connectionist Temporal Classification (CTC) can decode sequences of characters without explicit segmentation.

The probability of a transcription Y given input X is:

$$P(X/Y) = \sum_{\pi \in B^{-1}(Y)} P(\pi/X) \quad (7)$$

where, π represents all possible alignment paths.

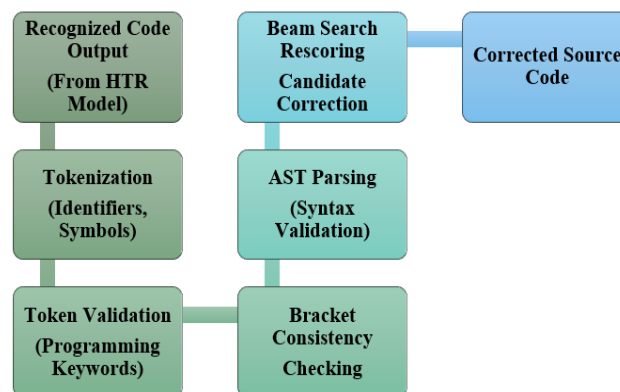


Figure 5. Syntax-aware error correction workflow for validating recognized code tokens.

tree (AST), which will validate the syntax of the code. Candidate corrections are produced if there is inconsistency, and these candidates are evaluated using a beam search to find the best syntactically correct code sequence.

Validation of Tokens: Tokens identified by the recognizer are validated against valid programming language tokens.

Fig. 5 illustrates the syntax-aware correction process after the recognition of code that has been identified.

First, after the recognition of the code, the text is tokenized into elements of a programming language (identifiers, keywords, etc.). These tokens are analyzed according to the syntax rules of the programming language used to identify the structural errors based on bracket consistency. Next, the recognized code is transformed into an abstract syntax

rescoring them using beam search that is directed by programming language models.

7. Algorithm: Syntax-Aware Recognition Pipeline

Algorithm 1: Syntax-Aware Recognition Pipeline

Algorithm 1: Syntax-Aware Recognition Pipeline

Input : Preprocessed handwritten code image I
Output: Corrected source code S
Data : Recognition model, token rules, syntax validation rules, beam search scorer, AST parser

```

 $L \leftarrow \text{SegmentLines}(I)$ ;
 $R \leftarrow []$ ;
foreach line  $l$  in  $L$  do
     $r \leftarrow \text{RecognizeWithCNNBiLSTMCTC}(l)$ ;
    Append  $r$  to  $R$ ;
 $T \leftarrow \text{Tokenize}(R)$ ;
Validate  $T$  using language rules;
Check bracket consistency in  $T$ ;
if  $T$  contains errors then
     $C \leftarrow \text{GenerateCandidateCorrections}(T)$ ;
     $T \leftarrow \text{SelectBestCandidateByBeamSearch}(C)$ ;
if  $\text{ASTParse}(T)$  is successful then
     $S \leftarrow T$ ;
else
     $S \leftarrow \text{BestRecoverableSequence}(T)$ ;
return  $S$ ;

```

a) Unicode Braille for reading by screen readers and displaying on digital screens

b) Braille-Ready Format (BRF) for use in Braille embossers

Braille representation of a program will preserve structure, allowing visually impaired students to read source code.

5. Experimental Setup

5.1. Dataset

For testing the proposed handwritten code recognition and Braille transcription system, a collection of handwritten programming code examples was developed for testing purposes. Forty students were each asked to write short programs using the most commonly used programming languages that are typically taught at schools. These included C, C++, Java, and Python. All forty students each created approximately twenty-five small programs with these programming languages. Small programs were examples of arithmetic operations,

Validation of Brackets: Use of a stack is used to validate parentheses and braces.

Validation of the Abstract Syntax Tree: After the corrected code has been parsed, it is used to create an abstract syntax tree (AST). If the parsing fails, then other sequences of tokens are generated.

Rescoring Candidate Sequences Using Beam Search: Candidate sequences are evaluated by

The syntax-aware recognition pipeline in Algorithm 1 transforms hand-written code images into valid source code by performing the following steps: Line Segmentation; Character Recognition Using a CNN-BiLSTM-CTC Model; Tokenization; Syntax Validation; Error Correction Using Beam Search; Final Verification Through Ast Parsing To Create Syntactically Valid Code.

4.8. Braille Output

Validated Source Code will be converted from source code into Braille with Unified English Braille (UEB) as the standard Braille computing convention.

Translation will include:

1. conversion of programming symbols to Braille characters
2. maintenance of indented lines
3. creation of Braille-ready files

Two types of output are created:

Variations of lighting and added lighting noise make it more difficult to recognize the programs and are representative of what students may experience when writing programs in a real-world environment.

Examples of the type of programs written by the students included programs written in C (with printf and scanf), C++ (with cin and cout), Java (with Scanner and System.out.println), and Python (with input() and print()).

conditional statements, input–output operations, etc.

Approximately one thousand handwritten code images were created from this data set. The common programming structures contained in the handwritten programs consisted of variable declaration, conditional statements, arithmetic expressions, function definitions, and input-output operations. Pages containing the student's hand-writing were photographed using a smartphone camera. Four lighting conditions were simulated during the photographing process. Lighting conditions included normal lighting, low light, shadow, and uneven lighting.

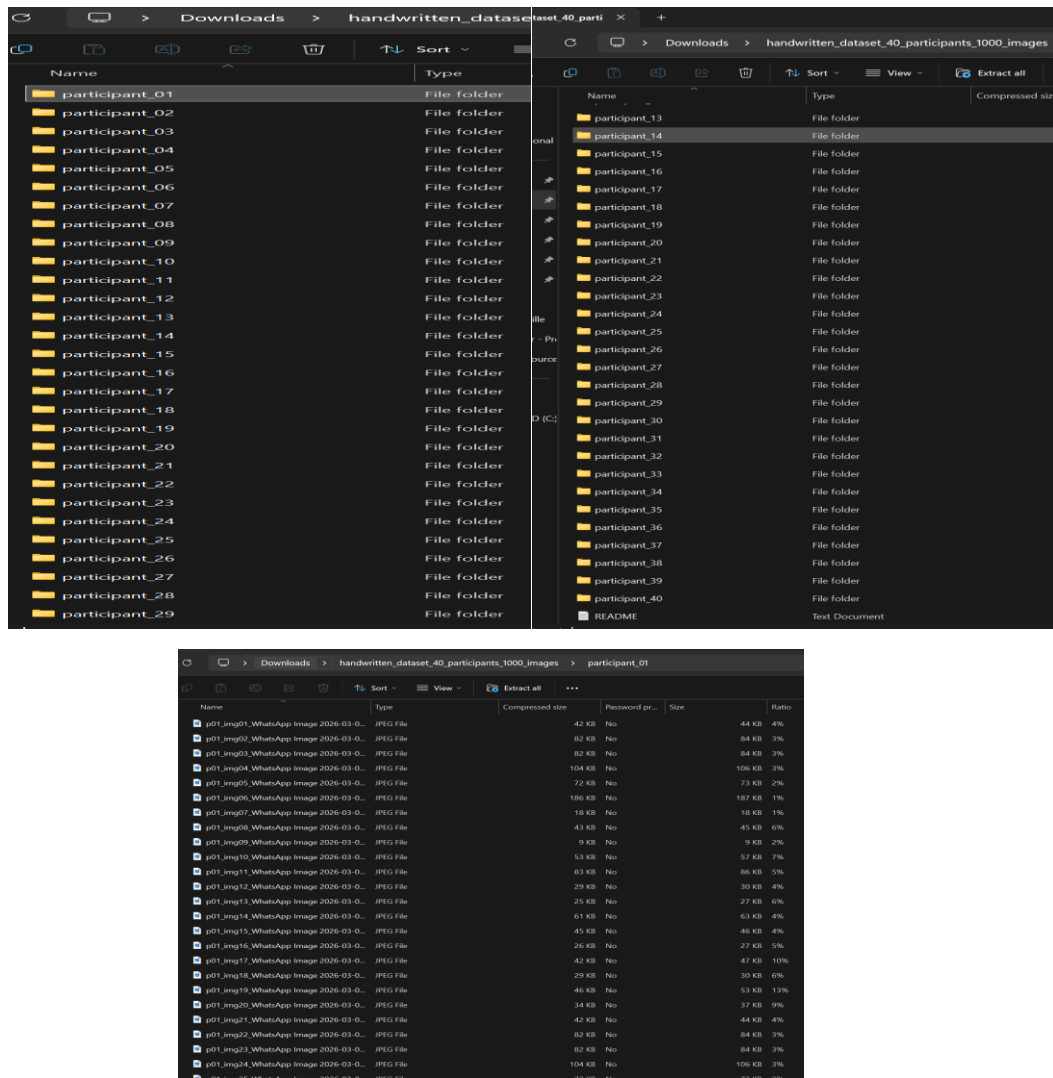


Figure 6. Handwritten Source Code Collected in the Dataset (C, C++, Java, Python)

ideal benchmark for assessing the robustness of a proposed handwritten code recognition system with Braille transcription.

The dataset consists of handwritten programming code from forty contributors. Each contributor produced twenty-five handwritten programming images, thus the dataset consisting of 1000 handwritten code images. The contributions consist of programming code examples in C, C++, Java, and Python that demonstrate common programming elements (arithmetic operators, conditional statements, loops, and input/output functions).

A representative example of the handwritten source code samples found in the dataset is presented in Fig. 6. Examples of handwritten source code in four programming languages (C, C++, Java, and Python) that represent common programming concepts (variable declarations, arithmetic operators, conditional statements, loops, function calls, input/output operations) are included in the dataset. The handwriting samples present variability in terms of writing style, spacing, symbols, indentation, and image quality, representing typical classroom and note-taking environments. This variability presents obstacles for handwritten code recognition systems, which make the dataset an

Table 1. Dataset characteristics and distribution

Participants	Programs per Participant	Total Images	Languages
40	25	1000	C, C++, Java, Python

5.2 Dataset Split

To ensure that the evaluation of the data set is robust and there will be no data leakage in the process of training and validating the model with the test set; The dataset was split at the participant level using a writer-independent protocol to prevent handwriting-style leakage between the training, validation, and test sets.

Table 2. Training and Hyperparameter Configuration

Dataset Split	Participant Count	Image Count	Percentage (%)	Primary Purpose
Train Set	28	700	70%	Deep Neural Network feature and sequence learning
Validation Set	4	100	10%	Hyperparameter tuning and early stopping trigger
Test Set	8	200	20%	Final evaluation of OCR,HTR, and SOTA Comparisons

5.3. Training Setup and Hyperparameters

Implementation of a handwritten code recognition module used Pytorch as well as the configurations for the CNN-BiLSTM-CTC recognizer model (image pre-processing parameters), decoder strategies, and syntactic validation tools are given in Table 3. It includes image normalization; Sauvola adaptive thresholding; convolutional extraction of features from images; bidirectional sequential modeling; CTC-based decoder; Beam Search rescored; Language specific Abstract Syntax Tree Parsing.

Table 3. Training Setup and Hyperparameter Configuration

Category	Parameter	Value / Setting
Input Specifications	Input Image Size	128 × 1024 pixels (grayscale)
Preprocessing	Sauvola Parameter k	0.2
Preprocessing	Dynamic Range R	128
Preprocessing	Sauvola Window Size	15 × 15
Architecture	CNN Backbone	7 convolutional layers
Architecture	Convolution Kernel	3 × 3
Architecture	Activation Function	ReLU
Sequence Modelling	BiLSTM Layers	2 bidirectional LSTM layers
Optimization	Optimizer	AdamW
Optimization	Learning-Rate Scheduler	Cosine annealing with warm restarts
Optimization	Batch Size	16 lines per batch
Optimization	Maximum Epochs	150
Optimization	Early Stopping	Triggered after 15 epochs without improvement
Decoding	CTC Blank Index	0
Decoding & Rescoring	Beam-Search Width	20 candidates
Syntax Validation	Python Parser	Python ast
Syntax Validation	C/C++ Parser	tree-sitter-c / tree-sitter-cpp
Syntax Validation	Java Parser	javalang

Validity Rate. The three evaluation metrics measure the degree of accuracy achieved by the proposed system's recognizer at different granularity levels, namely character, token, and syntax level, respectively.

1. Character Error Rate (CER)

Character Error Rate (CER) is an evaluation metric for measuring how many character recognition errors were made on character substitution (substitutions, deletions, and/or insertions), with respect to the number of characters in the ground truth sequence. In this context, CER is defined as follows:

$$CER = \frac{(S+D+I)}{N}$$

(8)

where S denotes the number of substitutions that occurred during the recognition process, D

The CNN (Convolutional Neural Network) extracts feature information related to space from the processed handwritten code images; meanwhile, the two layers of the Bidirectional Long Short-Term Memory (BiLSTM) model find temporal dependencies along the identified characters in the code. The CTC decoder can identify a sequence but does so as it does not have to know where one character ends and another begins. Following the decoding process, rescored candidates using a beam search method can be considered by a parser specific to that language's syntax.

5.4. Evaluation Metrics

To evaluate the performance of the proposed system, we used three evaluation metrics: Character Error Rate (CER), Token Accuracy, and Syntax

specific characteristics of the language itself. Therefore, Tesseract was used as the standard OCR comparative model to test how well a standard OCR approach could recognize handwritten programming-source code.

As previously stated, Tesseract recognizes source code written by hand without taking into consideration programming-language syntax, indentation, source code layout, or sequential constraints of source code.

2. Standard HTR Comparative Model (CRNN-CTC):

A Convolutional Recurrent Neural Network with Connectionist Temporal Classification (CRNN-CTC), which uses CTC decoding, was chosen as the comparative model for sequential recognition in handwritten text recognition. This model extracts features from images using convolutional layers and then uses a Bi-RNN decoder to decode the output of the previous layer. Because CRNN-CTC does not include programming-specific lexical validation, bracket consistency checks, beam-search rescoring, or AST-based syntax validation, it serves as a comparative model to the proposed framework.

3. Literature-based Reference Values (TrOCR):

Results from published studies of TrOCR-Small, TrOCR-Base, and TrOCR-Large on the IAM handwriting database were used as literature-based references for state-of-the-art transformer-based handwritten text recognition [31]. Like many transformer-based models [31], TrOCR uses pre-trained image and text transformers to perform end-to-end recognition. Although the IAM handwriting database was used in these referenced publications instead of the handwritten-programming-code data set used in this research, they have still been included here as literature-based reference values only and NOT as comparative reference values.

4. Proposed Model:

The proposed model includes multiple components, including image preprocessing, CNN-BiLSTM-CTC-based handwritten coding recognition, code reconstruction, lexical validation and

represents the number of deletions, I represent the number of insertions, and N represents the total number of characters in the ground truth. Therefore, if there are fewer substitutions, deletions, and/or insertions, then it will result in a lower CER, which in turn means that the proposed system's recognizer performs well.

2. Token Accuracy

Token accuracy is an evaluation metric for determining what percent of programming tokens from a ground truth code sample were correctly identified by the proposed system's recognizer. Tokens represent programming language elements such as identifiers, operators, literals, delimiters, and keywords. If there are more correct identifications of tokens, then there is higher token accuracy.

3. Syntax Validity Rate

Syntax Validity Rate evaluates the percentage of generated source-code examples which meet the syntax rules of their respective programming languages as validated through either parsing or syntax checking. This metric is critical since a transcript may contain minimal character error but potentially contain an erroneous symbol, delimiter or incorrect indentation making it syntactically incorrect.

5.5 Baseline Models for Comparison

In order to compare the performance of the proposed framework to other possible solutions, traditional Optical Character Recognition (OCR) and Handwritten Text Recognition (HTR) models were selected as comparative baselines. Additionally, published results from the Transformer-based TrOCR model were included as reference values based upon recent advances in handwritten text recognition to create an even larger context. Below is the description of each method that will be compared.

1. Standard OCR Comparative Model (Tesseract v5.3.0):

Traditional OCR has been widely accepted as being inadequate for recognizing handwritten source code because it doesn't take into account the

aware correction module improved performance by allowing the recognition system to validate common programming constructs such as conditional statements, function definitions, and bracket structures.

Additionally, the Braille transcription module has been shown to successfully convert recognized programming code into Braille representations while maintaining both the indentation structure and common programming symbols. Therefore, the visually impaired students can be able to read and understand programming code via Braille displays or printed embossed Braille paper.

bracket validation, beam-search rescoring, AST-based syntax validation, and Braille transcription. As opposed to traditional OCR and HTR models that do not use structural or syntactical information about the programming languages before creating their final accessible Braille representations: the proposed framework utilizes such programming-language-specific structural/syntactical information.

6. Results and Discussion

Experimental results show a substantial improvement in performance of the proposed handwritten code recognition system over baseline OCR and handwritten text recognition systems. The syntax-

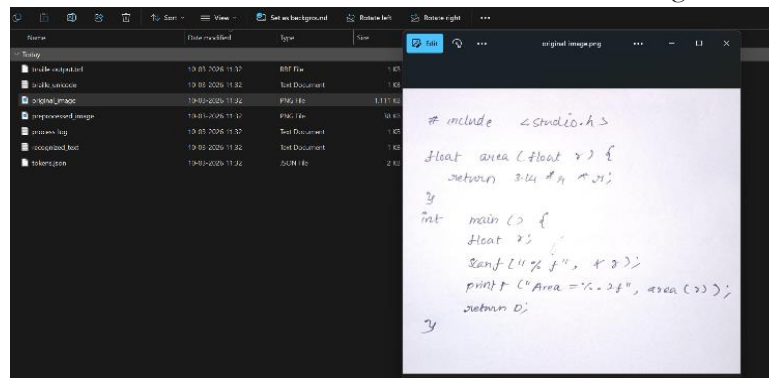


Figure 7. Handwritten programming code image (as input) for the proposed handwritten programming code recognition system.

participants and included natural variations in spacing, handwriting style, and how symbols are drawn.

The image of handwritten programming code in Fig. 7 was an example of the real handwritten programming notes that were collected from

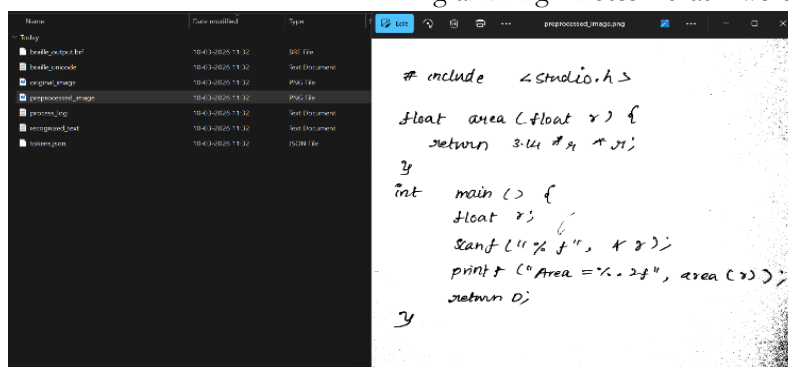


Figure 8. Handwritten code image that has been pre-processed

and a contrast improvement to enhance the readability of characters and eliminate background artifacts.

Preprocessing is used to make the handwritten code image better so it can be recognized. The preprocessing result is shown in Fig. 8 and includes the use of noise reduction, grayscale conversion,

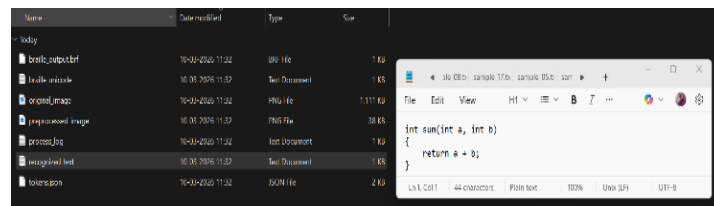


Figure 9. Source code text recognized by the handwriting recognition module.

The cleaned image after preprocessing is then passed to the handwriting recognition module, which uses a CNN-BiLSTM-CTC architecture. Fig. 9 shows the source code text that has been recognized by the model.

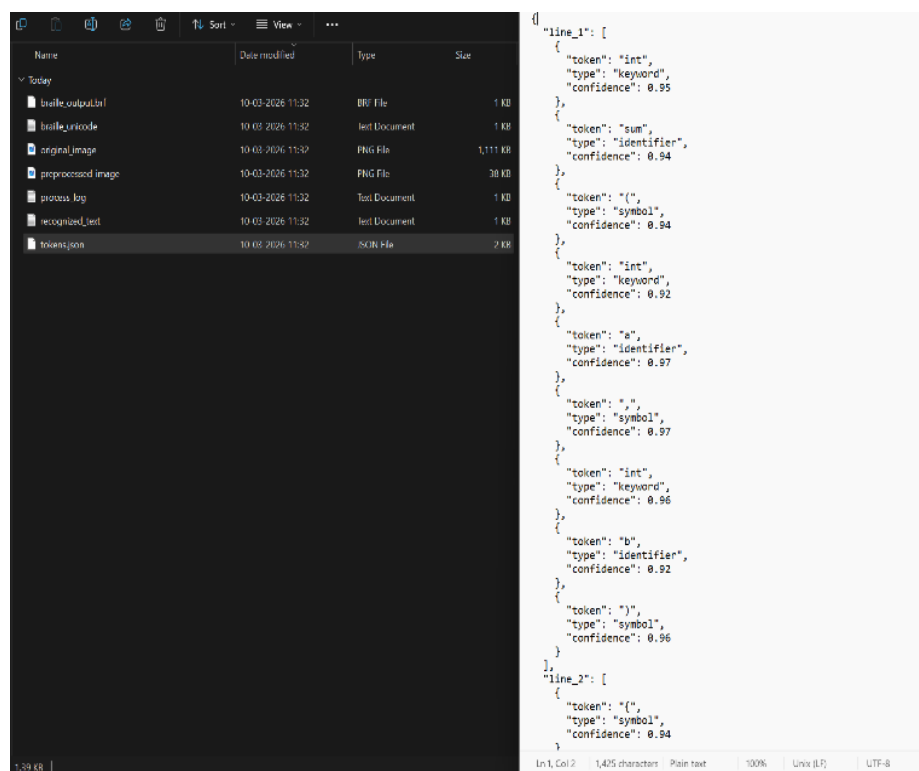


Figure 10. The recognized source code has been tokenised and converted to JSON format that contains confidence levels for each token.

of the source code and each token's corresponding confidence score. The token data is then formatted into JSON and displayed in Fig. 10.

Each token from the source code is then processed at the tokenisation level to extract the individual components (keywords, identifiers, operators, etc.)

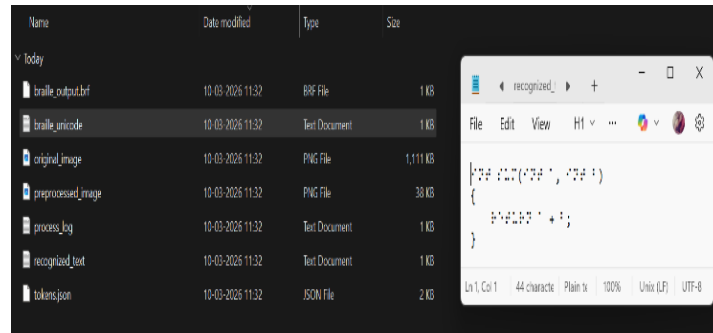


Figure 11. Unicode Braille transcription derived from the verified source code.

The Braille transcription that corresponds to the identified program is presented in Fig. 11.

The verified source code is converted to a Unicode Braille representation of it, according to the rules for computing Unified English Braille.

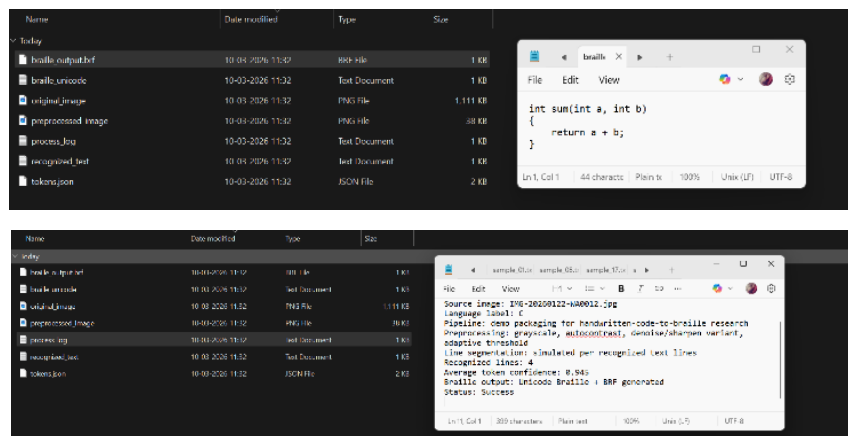


Figure 12. Braille Ready Format (BRF) file generated for Braille embosser devices.

a syntax validity rate. The proposed architecture was compared to conventional Optical Character Recognition (OCR) and Handwritten Text Recognition (HTR) architectures based on the same handwritten programming code data set. Additionally, results from published studies of Transformer-based TrOCR architectures on the IAM handwriting benchmark were used as reference values to better understand the current state of progress in the field of handwritten text recognition.

The final Braille transcription is exported in Braille Ready Format (BRF), which can be directly used by Braille embossers to produce tactile programming documents for visually impaired users, as shown in Fig. 12.

6.1 Overall Recognition Performance

The overall performance of character recognition by the proposed architecture was assessed using Character Error Rate (CER), Token Accuracy, and

Table 4. Overall Recognition Performance and Reference Comparison

Method	Evaluation Dataset	CER (%) ↓	Token Accuracy (%) ↑	Syntax Validity (%) ↑
OCR Baseline	Handwritten Programming Code	18.4	72.3	61.5
HTR Model	Handwritten Programming Code	11.2	84.7	76.9

TrOCR-Small [31]	IAM	4.22	—	—
TrOCR-Base [31]	IAM	3.42	—	—
TrOCR-Large [31]	IAM	2.89	—	—
Proposed Frame- work	Handwritten Programming Code	4.8	95.6	92.4

Our system achieved a CER of 4.8%, a token accuracy of 95.6%, and a syntax validity of 92.4%. The performance is representative of contributions from image preprocessing, CNN-BiLSTM-CTC recognition, code structure reconstruction, lexical and bracket validations, beam search rescoring, and AST-based syntax validations. In particular, the high level of syntax validity is very important for handwritten programming code recognition since errors involving operators, brackets, delimiters, or indentation can alter the program structure or create syntactically incorrect code. In addition, the inclusion of constraints related to the programming language increases the reliability of the reconstructed code while maintaining structural information necessary for Braille transcription.

Overall, the results indicate that recognizing sequential handwriting along with the structural and syntactical processes specifically associated with programming languages will provide a superior method for transcribing handwritten programming code compared to using traditional OCR or HTR methods.

6.2. Ablation Study

A module-by-module ablation study to determine the contributions of individual modules in the proposed system was performed. The Baseline Configuration (B0) is composed of the CNN-BiLSTM-CTC recognition architecture alone; it does not include enhancements to data preprocessing or syntactic analysis techniques that utilize the output of the recognizer. Beginning at B0, subsequent Configurations are created by successively adding: Illumination Correction and Contrast Limited Adaptive Histogram Equalization (CLAHE) (B1); Sauvola Thresholding and Deskewing (B2); Code Structure Recovery (B3); Lexical Validation and Bracket Validation (B4); Beam-Search

↓ A lower value of CER indicates good quality; an increase in token accuracy and syntax validity also represents good quality. Values for TrOCR were taken from literature (i.e., published results) on the IAM Benchmark and thus serve as literature-based reference values instead of a direct comparison to our experimental results. Conventional OCR had a CER of 18.4%, a token accuracy of 72.3%, and a syntax validity of 61.5%. These results show how well traditional OCR performs when recognizing hand-written programming codes. Variability in handwriting, specific symbols used in programs, indentations, and degradation of images all negatively impact recognition.

However, by using a conventional HTR model, we saw a great improvement in recognition performance. Specifically, the CER was reduced to 11.2%, and token accuracy and syntax validity were increased to 84.7% and 76.9%, respectively. This demonstrates that using sequence-based handwriting recognition has advantages over traditional OCR. Nevertheless, visual recognition alone can still generate misrecognized programming symbols or structurally incorrect code due to lack of explicit inclusion of programming-language syntax. TrOCR reported CERs for TrOCR-Small, TrOCR-Base, and TrOCR-Large on the IAM Benchmark to be 4.22%, 3.42%, and 2.89%, respectively [31]. These CERs are indicative of the success of transformer-based models for handwritten text recognition. Note that these benchmarks contain handwritten natural text. Handwritten programming codes have additional difficulties, including operators, brackets, delimiters, indentation, and language-specific syntax. Thus, we only use TrOCR's results as literature-based reference values. We do not treat them as direct experimental comparisons.

Rate (CER), Token Accuracy, and Syntax Validity, as well as Language-Specific Token Accuracy for C, C++, Java, and Python.

Rescoring (B5); and finally, the full system adds syntax validation via an Abstract Syntax Tree (AST). Each configuration's contribution was evaluated through three measures: Character Error

Table 5. Module-Wise Ablation Study Metrics Across Programming Languages

Model Variant / Configuration	Overall CER (%) ↓	Overall Token Accuracy (%) ↑	Overall Syntax Validity (%) ↑	C Token Accuracy (%)	C++ Token Accuracy (%)	Java Token Accuracy (%)	Python Token Accuracy (%)
B0: Baseline CNN-BiLSTM-CTC	18.2	71.4	60.1	72.8	70.2	68.5	74.1
B1: B0 + Illumination Correction + CLAHE	12.9	81.2	72.5	82.5	80.1	79.2	83.0
B2: B1 + Sauvola Thresholding + Deskewing	10.1	85.3	76.0	86.1	84.8	83.9	86.4
B3: B2 + Code Structure Recovery	8.8	87.6	80.2	87.9	86.5	85.8	90.2
B4: B3 + Lexical + Bracket Validation	7.3	88.4	84.1	89.2	87.9	87.1	89.4
B5: B4 + Beam-Search Rescoring	5.6	92.8	88.9	93.4	92.1	91.5	94.2
Full System: B5 + AST-Based Syntax Validation	4.8	95.6	92.4	96.2	95.1	94.8	96.3

Lower CER indicates better performance; higher token accuracy and syntax validity both indicate better configurations were also the same ones used in your current ablation analysis.

Incorporation of Sauvola thresholding and deskewing in B2 resulted in additional reductions in CER to 10.1% and increases in token accuracy to 85.3%. Incorporation of code-structure recovery in B3 produced a reduction in CER to 8.8% and a corresponding increase in syntax validity from 76.0% to 80.2%. A significant difference was observed for Python, as token accuracy increased from 86.4% to 90.2%, demonstrating the

performance.

CER of 18.2% , Token Accuracy of 71.4% and Syntax Validity of 60.1 % were obtained by the base-line configuration (b0). The addition of illumination corrections and CLAHE in B1 resulted in a CER value of 12.9% and token accuracy of 81.2%, which shows that normalization of illumination and local contrast enhances the quality of handwritten feature information provided to the recognition system. These B0 and B1

their own handwriting. There are many common confusions among numbers and identifiers like 0/O and 1/l/I, delimiters like () {} [] and programming operators/punctuation like /= and /;. If there are these types of confusion recognition errors, then they will create incorrect tokens, unmatched delimiters, or syntactically erroneous expressions. An example would be if a programmer has written a handwritten opening brace incorrectly, it will start to tokenize, but after that it will continue to give a delimiter mismatch error when checking the syntax. This is where rescoring via beam search could help find a better hypothesis for what was recognized by using some programming language constraints.

B. Structural/Spatial Disruption Errors

Most structural errors come about through slant (skew), irregular line spacing, drift in indentations, and poor separation of multi-line statements. Because of the use of indentation to denote blocks in Python, structural errors are very important for this language. For instance, if a handwritten line starts at different positions than others, then the handwritten lines will be incorrectly reconstructed into an indented block, which results in either an `IndentationError` or a `SyntaxError`. Likewise, if one statement is broken up over many lines without good indication of where the line continues, then the structural reconstruction will be done poorly. It is possible to have multiple plausible syntactical representations of the original structure, so the original structure may not always be able to be restored properly.

C. Compound Visual Noise Effecting Structure/ Syntax and Semantics

significance of maintaining indentation and line structure when recognizing indentation-sensitive programming languages.

Further decreases in CER to 7.3% and increases in Syntax Validity to 84.1% were realized through incorporation of lexical and bracket validations in b4. Further decreases in CER to 5.6% and increases in token accuracy to 92.8% were achieved through incorporation of beam search rescoring for B5, resulting in improvements in contextual decoding.

The Full System, combining both lexical constraints on candidates and AST-based syntax validations, demonstrated the lowest CER of 4.8%, along with an overall token accuracy of 95.6% and a syntax validity of 92.4%. Additionally, at the language level, token accuracy reached 96.3% for Python, 96.2% for C, 95.1% for C++, and 94.8% for Java.

In summary, the results of this ablation analysis demonstrate progressive improvements in performance as preprocessing, structural reconstruction, contextual decoding, and syntax-aware validation are each added individually to the system. As such, the decrease in CER from 18.2% in b0 to 4.8% in the full system demonstrates the cumulative contributions of all modules within the system.

6.3 Error Analysis and Failure Cases

Although the above proposed framework has demonstrated a significant improvement in the performance of recognizing programming characters, there are still some problems as follows: Handwriting programming symbols have strong visual similarities; Structural dependencies exist among them; There are also many different requirements for formatting based on programming languages. In order to analyze this limitation, the remaining recognition and syntax errors will be classified by their primary reason and whether they can be recovered using the framework.

6.3.1 Classification of Main Error Modes

A. Confusion in Symbols (recoverable errors)

Programming symbols that have been handwritten on paper may show a high degree of variation visually due to differences in how individuals write

enough information to parse successfully. These cases are more difficult to resolve since the error affects both visual interpretation of the code and the syntactic structure of the code.

6.3.2 Quantitative Distribution of Failures

To examine the frequency or quantity of the failures of AST-parsing observed during the experiments, each observed failure was classified into one of its main cause(s) and corresponding behavior upon recovery.

Compound errors happen when the visual noise of the image degrades the quality of multiple aspects of the program, including identifiers, operators, and delimiters. As an example, consider `for(int i = 0; i < n; i++)`. If the input had severe amounts of visual noise/smudges/low contrast in addition to missing required parentheses and semicolons, at least three ways to interpret this expression were available.

If enough structural symbols are lost simultaneously, the resultant sequence may not contain

Table 6. Quantitative Distribution and Recovery of Failure Cases

Failure Category	Primary Root Cause	Share of AST Failures (%)	System Recovery Status
Delimiter Mismatch	Ambiguous bracket or parenthesis drawing	38.5	Fully recovered via beam rescoring
Token Misclassification	Symbol overlap (1/l, ;/,, =/=)	29.2	Partially recovered via lexical rules
Indentation Misalignment	Handwriting drift or slant/skew	18.3	Fallback to best recoverable sequence
Missing Terminal Symbols	Omitted semicolons or trailing braces	9.0	Recovered via AST production rules
Unresolvable Noise	Extreme occlusion, smudging, or low contrast	5.0	Unrecoverable

These results indicate that many syntax failures arise due to visually ambiguous symbols and not necessarily due to total symbol recognition failure. A number of lexical-validation, bracket-checking, and beam-search rescoring techniques can help reduce errors arising from such ambiguities; however, the extreme case of severe degradation and structural ambiguity appears to be much harder to resolve.

The distributions show that delimiter mismatches are the most common type of AST-related failure (38.5%), followed by token misclassification as the second most frequent failure (29.2%). In third place is indentation misaligned (18.3%), with the fourth type being missing terminal symbols (9.0%), and lastly severely degraded input (5.0%), on which reliable recovery cannot be performed. These types and frequencies represent the current failure distribution of the manuscript.

incremental performance increases. Specifically, syntax-aware processing aided in reducing ambiguity of visually ambiguous programming symbols and identifying structurally incorrect sequences. Yet, severe degradation of images, missing symbols and poor indentation errors still persist when the original programme structure can't be correctly reconstructed.

In addition to recognising handwritten coding, generating Unicode Braille and BRF output also expands the scope of the framework to include content creation for visually impaired learners. The study suggests that combining recognition of handwriting with language constraints related to programming languages can increase the confidence in transcribing source code prior to converting it to braille. Future studies including large and varied data sets, new recognition architectures, and user-centred studies with visually impaired learners will enhance the generality and accessibility of the presented framework.

7. Conclusions

The primary goal of this research is to provide an extensive method of converting handwritten computer programs to Braille representations so that visually impaired students can be supported in their study of programming. The proposed system utilizes a combination of image preprocessing, handwritten code recognition, syntax-aware validation, and Braille transcriptions to convert handwritten source code images to machine-readable and Braille-accessible formats. Image preprocessing includes several processes, including illumination normalization, noise reduction, and contrast enhancement, to increase the image quality, which enables more accurate recognition of handwritten characters. Using a CNN-BiLSTM-CTC-based model, the system could translate the handwritten images of programming code into a format that preserves the structure of the code as well as the

6.3.3 Fallback Strategy for AST Errors That Cannot Recover

For unrecoverable AST errors, the system uses a fallback strategy that will allow it to gracefully degrade its operation. This is achieved through three mechanisms described below as part of BestRecoverableSequence in Algorithm 1.

- 1. Local Token Correction:** System retains corrections validated by lexical rules and bracket matching even if overall block parsing of AST fails.
- 2. Preservation of Braille Formatting:** Instead of stopping Braille generation, the system outputs partially corrected token sequence into Unicode Braille and BRF formats and appends an accessible warning flag ([Syntax Notice: Unverified Block]).
- 3. Feedback:** This ensures visually impaired users also receive readable code output and do not experience silent failure or loss of data completely.

6.4 Discussion

Handwritten programming-code recognition is improved through a combination of visual recognition and programming-specific structural and syntactic knowledge. Preprocessing images will remove the effects of illumination variance, noise and skewing on handwritten input data; furthermore, the CNN-BiLSTM-CTC model will recognise sequential handwritten code. Recognition alone does not ensure accurate preservation of both programming symbols and their structural relationships. For this reason, additional code structure recovery, lexical validation, beam search rescoring, and AST-based validation provide an improvement in the syntactical correctness of the reconstructed code.

Ablation studies demonstrated that each of the added modules to the framework provided

: A. Varshini; Supervision : S. Gopinathan; Project Administration : S. Gopinathan. All authors have reviewed and approved the final version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data set used in this study includes about 1000 handwritten programming codes in C, C++, Java, and Python, which are collected from 40 MSc computer science, MCAS, and MSc IT students of the Department of Computer Science, University of Madras, Chennai, India, with different lighting conditions for research purposes. The data related to the finding of the study will be provided on reasonable request by the corresponding author. Since the abovementioned dataset contains the handwritten sample collected especially for this research, it is not available online, and it will be used continuously in future research and for academic purposes.

Acknowledgments: Authors would like to extend their special thanks to the Department of Computer Science, University of Madras, Chennai, India, for providing them the required academic support and research facility to do this work. Authors also thank those participants who volunteered to contribute handwritten programming samples for use in creating the dataset for this study.

Conflicts of Interest: There is no conflict of interest declared by authors since there is no influence on this paper either financially, personally, or institutionally. Authors confirm that no financial or institutional relationship has influenced the work reported in this manuscript. The research did not receive any outside funding; therefore, the funder had no part in the design of the study, in the collection, analysis, and/or interpretation of data, in the writing of the manuscript, or in the decision to publish the results.

References

- A. Graves, M. Liwicki, S. Fernández, R. Bertolami, H. Bunke, and J. Schmidhuber, "A novel connectionist system for unconstrained handwriting recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 5, pp. 855–868, 2009. doi: 10.1109/TPAMI.2008.137.
- A. Graves and J. Schmidhuber, "Offline handwriting recognition with multidimensional recurrent neural networks," in *Advances in Neural Information Processing Systems 21 (NeurIPS 2008)*, 2008, pp. 545–552. Available: <https://proceedings.neurips.cc/paper/2008>

actual textual code. To further improve the reliability of the system, a syntax-aware error correction module was integrated to validate the recognized tokens, check the consistency of brackets, and evaluate the structure of the program. The corrected valid code was then converted into Braille-ready format (BRF) and Unicode Braille formats to enable visually impaired programmers to view their coding content via Braille display and embosser systems.

Experiments were conducted using a dataset of handwritten programming examples collected from several individuals to demonstrate that the proposed system achieved significant improvements in recognition accuracy and syntactic correctness when compared to baseline Optical Character Recognition (OCR) methods for transcribing handwritten programming code. Overall, the proposed system provides a practical solution to bridge the programming education accessibility gap by providing a means for automatically translating handwritten programming code into Braille formats. Future development plans include expanding the system to accommodate additional programming languages, improving the accuracy of recognition of handwritten code written in complex handwriting styles, and developing the system to be integrated into mobile and assistive technology-based learning environments for real-time programming content accessibility.

Author Contributions: Conceptualization : A. Varshini, S. Gopinathan; Methodology : A. Varshini; Software : A. Varshini; Validation : A. Varshini, S. Gopinathan; Formal analysis : A. Varshini; Investigation : A. Varshini; Resources : S. Gopinathan; Data curation : A. Varshini; Writing — original draft preparation : A. Varshini; Writing — review & Editing : S. Gopinathan; Visualization

- A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, "Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks," in *Proceedings of the 23rd International Conference on Machine Learning (ICML)*, 2006, pp. 369–376. <https://dl.acm.org/doi/10.1145/1143844.1143891>
- T. Bluche, "Joint line segmentation and transcription for end-to-end handwritten paragraph recognition," in *Advances in Neural Information Processing Systems 29 (NeurIPS 2016)*, 2016, pp. 838–846. <https://proceedings.neurips.cc/paper/2016>
- J. Puigcerver, "Are multidimensional recurrent layers really necessary for handwritten text recognition?" in *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, vol. 1, 2017, pp. 67–72. doi: 10.1109/ICDAR.2017.20.
- V. Pham, T. Bluche, C. Kermorvant, and J. Louradour, "Dropout improves recurrent neural networks for handwriting recognition," in *2014 14th International Conference on Frontiers in Handwriting Recognition (ICFHR)*, 2014, pp. 285–290. doi: 10.1109/ICFHR.2014.55.
- A. Poznanski and L. Wolf, "CNN-N-Gram for handwriting word recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2305–2314. <https://ieeexplore.ieee.org/document/7780635>
- B. Moysset and R. Messina, "Are 2D-LSTM really dead for offline text recognition?" *International Journal on Document Analysis and Recognition*, vol. 22, no. 3, pp. 193–208, 2019. doi: 10.1007/s10032-019-00325-0.
- J. Michael, R. Labahn, T. Grüning, and J. Zöllner, "Evaluating sequence-to-sequence models for handwritten text recognition," in *2019 International Conference on Document Analysis and Recognition (ICDAR)*, 2019, pp. 1286–1293. doi: 10.1109/ICDAR.2019.00208.
- A. F. de Sousa Neto, B. L. D. Bezerra, G. C. de Moura, and A. H. Toselli, "Data augmentation for offline handwritten text recognition: A systematic literature review," *SN Computer Science*, vol. 5, art. 258, 2024. doi: 10.1007/s42979-023-02583-6.
- R. Hussain, A. Raza, I. Siddiqi, K. Khurshid, and C. Djeddi, "A comprehensive survey of handwritten document benchmarks: Structure, usage and evaluation," *EURASIP Journal on Image and Video Processing*, vol. 2015, art. 46, 2015. doi: 10.1186/s13640-015-0102-5.
- K. Nikolaidou, M. Seuret, H. Mokayed, and M. Liwicki, "A survey of historical document image datasets," *International Journal on Document Analysis and Recognition*, vol. 25, pp. 305–338, 2022. doi: 10.1007/s10032-022-00405-8.
- S. S. Singh and S. Karayev, "Full page handwriting recognition via image to sequence extraction," in *Document Analysis and Recognition – ICDAR 2021, Lecture Notes in Computer Science*, vol. 12823, 2021, pp. 55–69. doi: 10.1007/978-3-030-86334-0_4.
- F. Wolf and G. A. Fink, "Self-training for handwritten word recognition and retrieval," *International Journal on Document Analysis and Recognition*, vol. 27, pp. 225–244, 2024. doi: 10.1007/s10032-024-00484-9.
- A. Khormi, M. Alahmadi, and S. Haiduc, "A study on the accuracy of OCR engines for source code transcription from programming screencasts," in *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*, 2020, pp. 65–75. doi: 10.1145/3379597.3387468.
- J. Perianez-Pascual, R. Rodríguez-Echeverría, L. Burgueño, and J. Cabot, "Towards the optical character recognition of DSLs," in *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE '20)*, 2020. doi: 10.1145/3426425.3426937.
- A. Mountapmbeme, O. Okafor, and S. Ludi, "Addressing accessibility barriers in programming for people with visual impairments: A literature review," *ACM Transactions on Accessible Computing*, vol. 15, no. 1, art. 7, 2022. doi: 10.1145/3507469.
- J. Seo and M. Rogge, "Coding non-visually in Visual Studio Code: Collaboration towards accessible development environment for blind programmers," in *Proceedings of the 25th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '23)*, 2023, pp. 102:1–102:9. doi: 10.1145/3597638.3614550.
- K. Albusays, S. Ludi, and M. Huenerfauth, "Interviews and observation of blind software developers at work to understand code navigation challenges," in *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '17)*, 2017, pp. 91–100. doi: 10.1145/3132525.3132550.

-
- C. M. Baker, C. L. Bennett, and R. E. Ladner, "Educational experiences of blind programmers," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*, 2019, pp. 759–765. doi: 10.1145/3287324.3287410.
- C. M. Baker, L. R. Milne, and R. E. Ladner, "StructJumper: A tool to help blind programmers navigate and understand the structure of code," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*, 2015, pp. 3043–3052. doi: 10.1145/2702123.2702589.
- E. Schanzer, S. Bahram, and S. Krishnamurthi, "Accessible AST-based programming for visually-impaired programmers," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*, 2019, pp. 773–779. doi: 10.1145/3287324.3287499.
- E. Schanzer, S. Bahram, and S. Krishnamurthi, "Adapting student IDEs for blind programmers," in *Proceedings of the 20th Koli Calling International Conference on Computing Education Research (Koli Calling '20)*, 2020. doi: 10.1145/3428029.3428051.
- O. Falase, A. F. Siu, and S. Follmer, "Tactile code skimmer: A tool to help blind programmers feel the structure of code," in *Proceedings of the 21st International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '19)*, 2019, pp. 536–538. doi: 10.1145/3308561.3354616.
- M. Ehtesham-Ul-Haque, S. M. Monsur, and S. M. Billah, "Grid-Coding: An accessible, efficient, and structured coding paradigm for blind and low-vision programmers," in *Adjunct Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology (UIST Adjunct '22)*, 2022. doi: 10.1145/3526113.3545620.
- A. C. Smith, J. M. Francioni, and S. D. Matzek, "A Java programming tool for students with visual disabilities," in *Proceedings of the 4th International ACM Conference on Assistive Technologies (Assets '00)*, 2000, pp. 142–148. doi: 10.1145/354324.354356.
- W. Schweikhardt, "A programming environment for blind APL-programmers," in *Proceedings of the International Conference on APL (APL '82)*, 1982, pp. 325–331. <https://dl.acm.org>
- J. A. Ramirez-Orta, E. Xamena, A. G. Maguitman, E. Milios, and A. J. Soto, "Post-OCR document correction with large ensembles of character sequence-to-sequence models," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 10, pp. 11192–11199, 2022. doi: 10.1609/aaai.v36i10.21369.
- Y. Bassil and M. Alwani, "OCR context-sensitive error correction based on Google Web 1T 5-Gram data set," *arXiv preprint arXiv:1204.0188*, 2012. Available: <https://arxiv.org/abs/1204.0188>
- J. Guerreiro, D. Gonçalves, D. Marques, T. Guerreiro, H. Nicolau, and K. Montague, "The today and tomorrow of Braille learning," in *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '13)*, 2013, art. 71. doi: 10.1145/2513383.2513415.
- M. Li, T. Lv, J. Chen, L. Cui, Y. Lu, D. Florencio, C. Zhang, Z. Li, and F. Wei, "TrOCR: Transformer-Based Optical Character Recognition with Pre-trained Models," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 11, pp. 13094–13102, 2023, doi: 10.1609/aaai.v37i11.26538.
-