



BINUS UNIVERSITY

BINUS INTERNATIONAL

<p>Braille Translator using Parser and Lexer</p>

Student Information:

Surname:	Given Name:	Student ID Number:
Kusnadi	Clarissa Audrey Fabiola	2602118490
Aditama	Galih Putra Aditama	2602227421
Emirryan	Muhammad Harmain Asyi Emir	2602206770
Sutiono	Rafael Sutiono	2602174535

Course Code : COMP6062001 **Course Name :** Compilation Techniques

Class : L5AC **Lecturer :** Ir. Tri Asih Budiono, M.I.T.

Type of Assignment: Final Project Report

Table of Content

I. Introduction.....	3
A. Background.....	3
B. Problem Description.....	3
C. Aim (Goal).....	4
II. Related Works.....	4
III. Implementation.....	5
A. Formal Description of the Computational Problem.....	5
B. Design of the Compiler (or Translator).....	5
C. Complexity Analysis for Algorithms and Data Structures Involved.....	7
D. Specification of Token (Regular Expression).....	7
E. Specification of Grammar (Context Free Grammar).....	8
IV. Evaluation.....	9
A. Implementation Details.....	9
B. Datasets.....	16
C. Experimental (Test Run) Analysis and Results.....	16
V. Discussions.....	18
VII. Program Manual (Screenshots).....	19
A. Application Interface.....	19
B. How to Use the Braille Translator Application.....	19
IX. References.....	22

I. Introduction

A. Background

Braille is a tactile writing system specially designed to aid people who are blind or have visual disabilities in their reading and writing activities. Invented in 1824 by Louis Braille when he was only 15 years old, this system is based on the French alphabet and is an improvement upon an earlier predecessor called Night Writing invented by Charles Barbier de la Serre. The original Braille system was published in 1829, including musical notation, and was later revised in 1837 to become the first binary writing system of the modern era (Jiménez et al., 2009). In the Braille system, a 3x2 matrix of raised dots is used; this is called a Braille cell. In this cell, the particular arrangement of dots uniquely represents each character, number, or punctuation mark. An example of a sentence represented in Braille. The sentence “I’m about to make a name for myself right here” in braille is.

ᠤᠨ ᠪᠣᠭᠡᠢ ᠵᠢᠰᠦᠨ ᠶ᠋ᠢᠨ ᠲᠤᠨ ᠬᠡᠮᠴᠢᠨ ᠳᠤᠨ ᠶ᠋ᠢᠨ ᠲᠤᠨ ᠬᠡᠮᠴᠢᠨ

Braille is an essential tool for the literacy and independence of the visually impaired, permitting them to achieve access to education, information, and public infrastructures. Still, despite the importance of braille, few tools exist that are both effective and accurate to translate text and braille, creating a gap between technology and awareness. This project aims to bridge that gap by developing a desktop application that translates English text to Braille and vice versa, fostering a broader understanding of Braille's importance in society. The desktop application will implement the lesson of Compilation Techniques in using Lexer and Parser for translation.

Lexical analysis is defined as the first process which occurs during the compilation process. This is done by breaking down the code into individual tokens. Tokens are the smallest possible individual unit in terms of programming (GeeksforGeeks, 2024a). While the lexical analysis breaks down the inputted code parsing is the process of analyzing the token order which would determine the synthetic structure of the token. The token stream would later be processed into a parse or syntax tree. The created tree would provide a hierarchical structure that would represent the program syntax. This is important due to the next step of the code generation and execution (GeeksforGeeks, 2024b).

The proposed project aims to produce a desktop application using Lexer and Parser as the translator, that is accessible to the general community with the hope of raising awareness and improving inclusivity.

B. Problem Description

Although Braille is an important part of the lives of blind and visually impaired people, there remains a lack of accessible, user-friendly tools for converting between Braille and English text. These gaps limit the spread and use of Braille among the general public and, ultimately, lower the possibility of achieving the goal of an inclusive society wherein people with visual

impairments can fully participate. Moreover, Braille translation is more than a one-to-one mapping of characters. Grade 2 Braille, for example, features contractions and abbreviations that make translation complex. Therefore, a strong solution needs to contain a good implementation and understanding of parsing techniques to translate such variations correctly.

C. Aim (Goal)

The aim of this project is to help in the translation of English (U.S.) level 2 Braille to the focus language of English as the more commonly used language and vice versa. This is done by creating a translator from English sentences to English (U.S.) level 2 Braille, ensuring seamless translation between the two languages. A desktop application will be created for it, allowing users to translate Braille to text, and the other way around.

Through the Braille translator application, we aim to provide a platform for users to be able to learn basic Braille in ease. The goal of this project is not to replace other formats of teaching Braille, such as the app visual brailier and online courses which could result in a better understanding of Braille to the user. Instead, the project focuses on giving users a basic and general understanding of the language. This would help in spreading awareness to the importance of learning what those who are visually impaired have to go through to the general public.

Furthermore, the project also aims to implement a lexer and parser process in translating English sentences to Braille. The goal of the app would be to be able to tokenize the sentence into individual tokens, which then the parser would be able to generate a parser tree based on the predefined grammar rule. After this process is done, the app would map it back to the requested Braille translation, ensuring a smooth process of the Braille translator.

II. Related Works

The main inspiration for this project is the work by Nahar et al. (2019) conducted a usability evaluation of "MBRAILLE," a mobile phone-based Braille learning application. Their findings indicate that such applications can effectively enhance Braille literacy and engagement among users. While this research highlights successful mobile implementations, our project aims to further accessibility through a desktop application designed specifically for educational use, allowing for a more detailed and comprehensive learning environment.

And in another study, Nahar et al. (2015) focused on the design of a Braille learning application tailored for visually impaired students in Bangladesh. Their research outlines principles for creating educational technologies that meet the needs of this demographic. Their project builds on these principles by combining both translation and education, ensuring that users not only translate text but also learn the fundamentals of Braille in an interactive manner.

These attempts often lack a user-friendly design and target specific groups of people. By employing both back-to-back braille and text translations, our project is a user-friendly and easy way to translate both braille and text for all groups of people, whether it be for learning, translation, or for other uses.

III. Implementation

A. Formal Description of the Computational Problem

The computational problem is to design a system which will parse sentence in a natural English language and check them for syntactical correctness in relation to the a prior defined CFG, performing further the operation of translation between English text and Braille, the solution must handle multi-digit numbers and tokenization of proper English sentences and translate back from Braille into English text perfectly.

B. Design of the Compiler (or Translator)

The design of this Braille Translator system follows the typical structure of a compiler or translator in which the input text, in English, undergoes several stages of processing to produce the equivalent Braille output. The system follows the steps of a traditional compilation process for translating a proper English sentence to Braille: Lexical Analysis, Syntax Analysis (Parsing), and Translation.

1. Input: Proper English sentence

The input to the system is a proper English sentence that follows the provided specified grammar rule. It may contain words and multi-digit numbers. For example: 'The cat will not eat its food'

2. Lexical Analysis (Lexer)

The first step of input processing is lexical analysis, where it tokenizes the input English text. The lexer, or tokenizer, breaks the raw input string into a sequence of tokens that can be further processed.

Example Tokenization: For the input 'The cat will not eat its food', the lexer produces the following tokens:

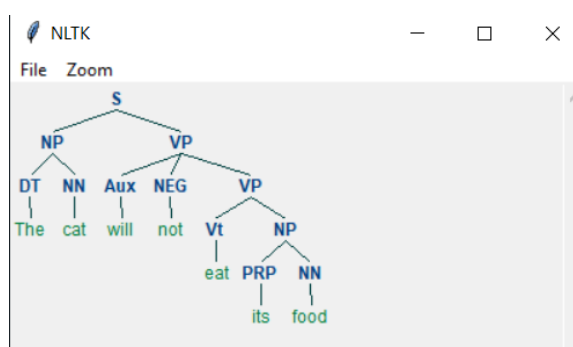
['The', 'cat', 'will', 'not', 'eat', 'its', 'food']

3. Parsing

After the input has been tokenized, the next step is parsing, which is the analysis of the syntactical structure of the sentence based on predefined grammar rules. The parser follows those rules to structure the tokens into something called a parse tree. The parse tree will represent the grammatical structure of the sentence, where the nodes of the tree will either represent a token, a phrase, or a syntactic element, for example, subject, verb, and object.

For example, the sentence 'The cat will not eat its food' could be parsed as:

(S
(NP (DT The) (NN cat))
(VP (Aux will) (NEG not) (VP (Vt eat) (NP (PRP its) (NN food))))))



4. Grammar (Context Free Grammar - CFG)

The system uses CFG in specifying the syntactic rules for English to Braille translation. The grammar ensures that every sentence is a proper English sentence. The CFG that we have created focuses on grammar rules for generating proper English sentences. It consists of rules for Sentence (S), Verb Phrase (VP), Noun Phrase (NP), Prepositional Phrase (PP), Preposition (PRE), and Number (NUM).

5. Token Mapping to Braille

Having parsed the sentence and generated a valid parse tree, the next step is to translate the parsed tokens into their corresponding Braille symbols. The translation is performed with respect to the Grade 2 Braille system, which involves both direct character mappings and contraction handling.

The system has a Braille Mapping Table in which every English token, be it a letter, number, or punctuation, maps to a corresponding Braille symbol.

For example, the sentence ‘The cat will not eat its food’ could be mapped as:

Word-level contraction found: 'The' -> '∴'

Letter match: 'c' -> 'ʔ'
Letter match: 'a' -> 'ʔ'
Letter match: 't' -> 'ʔ'
Letter match: 'w' -> 'ʔ'
Letter match: 'i' -> 'ʔ'
Letter match: 'l' -> 'ʔ'
Letter match: 'l' -> 'ʔ'
Letter match: 'n' -> 'ʔ'
Letter match: 'o' -> 'ʔ'
Letter match: 't' -> 'ʔ'
Letter match: 'e' -> 'ʔ'
Letter match: 'a' -> 'ʔ'
Letter match: 't' -> 'ʔ'
Letter match: 't' -> 'ʔ'
Letter match: 's' -> 'ʔ'
Letter match: 'f' -> 'ʔ'
Letter match: 'o' -> 'ʔ'
Letter match: 'o' -> 'ʔ'
Letter match: 'd' -> 'ʔ'

6. Generate Output: Braille Text

Once the tokens have been mapped to the appropriate Braille symbols, the system will generate the final output of the translated English sentence in Braille.

For example, the sentence ‘The cat will not eat its food’ have the output of:

Grade 2 Braille Translation:

Grade 2 Braille translation:

Meanwhile, for translation of Braille to English, the system does not need to go through the hassle of Lexer and Parser. It immediately maps the Braille symbol as a Grade 1 Braille to corresponding letters in English or if the Braille is a Grade 2, then it maps it to the corresponding words in English.

For example, for the Braille input of ‘ $\therefore \overset{1}{\cdot} \overset{2}{\cdot} \overset{3}{\cdot} \overset{4}{\cdot} \overset{5}{\cdot} \overset{6}{\cdot} \overset{7}{\cdot} \overset{8}{\cdot} \overset{9}{\cdot} \overset{0}{\cdot}$ ’,

The system will first find if there are any Grade 2 Braille text. If there are any, the output will be like follow,

Word-level contraction found: ':' -> 'the'

After that, the system will map the rest of the corresponding Braille symbols to the English tokens, and reconstruct the English sentence.

An example output will be like follow, 'Translated Text: the cat will not eat food'

C. Complexity Analysis for Algorithms and Data Structures Involved

Time complexity:

- Translation to Braille: $O(n)$, where n is the combined length of the input text, the number of words, and the average number of contractions checked per word.
- Translation from Braille: $O(n)$, where n is the length of the Braille text, m is the number of words, and k is the number of characters or contractions checked per word.
- Grammar construction: $O(n)$, it involves mapping non-terminals to lists of terminal words.
- Parsing sentences: $O(n^3)$, the chart parser used by NLTK comes from the CYK algorithm, which is used for determining whether a given string can be generated by the CFG and to draw the parse tree(s).

Overall time complexity: $O(n^3)$

Space complexity:

- $O(n+k+d)$, where:
 - n is the length of the input string.
 - k is the length of the output (Braille or English).
 - d is the total size of the dictionaries ($O(n_1+n_2+n_3)$), where n_1 , n_2 , n_3 are the sizes of GRADE1_BRaille, BRaille_NUMBERS, and GRADE2_CONTRACTIONS).

D. Specification of Token (Regular Expression)

To specify the tokens used in the Braille translation program, different types of English word classes are listed in the format of a text (.txt) file. This includes:

- Adjective (Adj): Words that describe a noun or pronoun. E.g. big, pretty, boring, etc.
- Adverb (Adv): Words that modify a verb. E.g. quickly, slowly, rarely, etc.
- Auxiliary Verb (Aux): Helping verbs that support the main verb in a sentence. E.g. do, have, will, can, etc.
- Conjunction (CONJ): Words that link other sentences, phrases, or words together. E.g. and, or, but, etc.
- Determiner (DT): Words that come before nouns to specify which, how many, or whose noun is being referred to. E.g. the, a, an, etc.
- Preposition (PRE): Words that indicate the location, position, direction, time, or manner of an action or event. E.g. in, at, on, of, etc.
- Negation (NEG): Words that make a sentence negative. E.g. not, etc.
- Noun (NN): Words that name something. E.g. cat, dog, etc.
- Number (NUM): An arithmetical value used to represent quantity. E.g. 1, 2, 3, etc.

- Pronoun (PRP): Words used to replace a noun. E.g. she, I, he, etc.
- Intransitive Verb (Vi): Verbs that do not require a direct object to act upon. E.g. yawn, sit, vote, etc.
- Transitive Verb (Vt): Verbs that require a direct object to act upon. E.g. borrow, pay, bring, etc.

Numerous words according to their English Word classes are entered into a text file according to their category.

≡ Adj.txt
≡ Adv.txt
≡ Axx.txt
≡ CONJ.txt
≡ DT.txt
≡ IN.txt
≡ NEG.txt
≡ NN.txt
≡ NUM.txt
≡ PRP.txt
≡ Vi.txt
≡ Vt.txt

The input sentences will be tokenized, in which the tokens will be used to map words in sentences according to their respective word classes. The tokens will also be used in the Context Free Grammar (CFG) rules specified for the program to construct parse trees and generate the Braille translation.

E. Specification of Grammar (Context Free Grammar)

The CFG that we have created focuses on grammar rules for generating proper English sentences. It consists of rules for Sentence (S), Verb Phrase (VP), Noun Phrase (NP), Prepositional Phrase (PP), Preposition (PRE), and Number (NUM).

Start Rule

- S (Sentence): The starting non-terminal that represents a proper English sentence.

Grammar Rules

1. Sentence (S):

- S → NP VP (A sentence can consist of a noun phrase followed by a verb phrase. E.g. The cat sleeps.)
- S → NP (A noun phrase can independently act as a sentence. E.g. The dog)
- S → VP (A verb phrase can independently act as a sentence. E.g. Runs quickly.)
- S → NUM (A number can independently act as a sentence. E.g. 10)
- S → S CONJ S (Sentences can be joined using conjunctions. E.g. The dog barks and the cat sleeps.)

2. Verb Phrase (VP):

- VP → Vi (An intransitive verb can independently act as a verb phrase. E.g. Runs.)
- VP → Vt (A transitive verb can independently act as a verb phrase. E.g. Eats.)
- VP → Vt NP (A verb phrase can consist of a transitive verb followed by a noun phrase. E.g. Eat food.)
- VP → Vt Adv (A verb phrase can consist of a transitive verb followed by an adverb. E.g. Runs quickly.)
- VP → Aux VP (A verb phrase can consist of an auxiliary verb followed by a verb phrase. E.g. Will run.)

- VP -> Adj VP (A verb phrase can consist of an adjective followed by a verb phrase. E.g. Quickly runs.)
- VP -> Aux NEG VP (A verb phrase can consist of an auxiliary followed by negation, then a verb phrase. E.g. Did not run.)
- VP -> Vt PP (A verb phrase can consist of a transitive verb followed by a preposition. E.g. Eat food on table.)
- VP -> VP CONJ VP (A verb phrase can consist of a verb phrase followed by a conjunction, then a verb phrase again. E.g. Eat and run.)
- VP -> VP NEG VP (A verb phrase can consist of a verb phrase followed by a negation, then a verb phrase again. E.g. Runs but not jumps.)
- VP -> PRE VP (A verb phrase can consist of a preposition followed by a verb phrase. E.g. To run.)

3. Noun Phrase (NP):

- NP -> DT NN (A noun phrase can consist of a determiner followed by a noun. E.g. The dog.)
- NP -> NP PP (A noun phrase can consist of a noun phrase followed by a prepositional phrase. E.g. The cat at home.)
- NP -> NN (A noun can independently act as a noun phrase. E.g. Dog.)
- NP -> NUM NN (A noun phrase can consist of a number followed by a noun. E.g. 3 cats.)
- NP -> NP CONJ NP (A noun phrase can consist of a noun phrase followed by a conjunction, then a noun phrase again. E.g. Cat and dog.)
- NP -> Adj NN (A noun phrase can consist of an adjective followed by a noun. E.g. Pretty cat.)
- NP -> DT Adj NN (A noun phrase can consist of a determiner followed by an adjective, then a noun. E.g. The big cat.)
- NP -> NP Aux PRP (A noun phrase can consist of a noun phrase followed by an auxiliary verb, then a pronoun. E.g. The book is mine.)
- NP -> PRP NN (A noun phrase can consist of a pronoun followed by a noun. E.g. Her cat.)

4. Prepositional Phrase (PP):

- PP -> PRE NP (A prepositional phrase can consist of a preposition followed by a noun phrase. E.g. On the table.)
- PP -> PRE VP (A prepositional phrase can consist of a preposition followed by a verb phrase. E.g. Before eating.)

5. Number (NUM):

- NUM -> NUM NUM (Numbers can concatenate to form numbers with more than one digit. E.g. 123)

IV. Evaluation

A. Implementation Details

The project was built using the Python programming language. Libraries in Python that will be used for the project includes:

- **OS:** A module with functions that interacts with the operating system. Used to open and read the token text files.
- **Tkinter:** A library used to construct basic Graphical User Interface (GUI) applications. Used to build the whole GUI for our Braille translator and to render the GUI of the generated parse tree.
- **Nltk:** A library used for Natural Language Processing (NLP). Used to define and utilize specific CFG. Moreover, it parses the sentences and generates parse trees for them.
- **Re:** A module that provides regular expression matching operations. Used to preprocess text to ensure sentences are properly tokenized.

Code Implementation

```
GRADE1_BRaille = {
    # Alphabets
    'a': '⠁', 'b': '⠃', 'c': '⠉', 'd': '⠙', 'e': '⠑',
    'f': '⠋', 'g': '⠗', 'h': '⠓', 'i': '⠏', 'j': '⠛',
    'k': '⠅', 'l': '⠊', 'm': '⠍', 'n': '⠝', 'o': '⠕',
    'p': '⠞', 'q': '⠟', 'r': '⠗', 's': '⠎', 't': '⠞',
    'u': '⠥', 'v': '⠺', 'w': '⠠', 'x': '⠭', 'y': '⠽',
    'z': '⠵',
    ' ': '⠆',
}
```

A dictionary mapping lowercase English letters and spaces to their Grade 1 Braille equivalents. This is the base for transcribing single characters to Grade 1 Braille.

```
CAPITAL_SIGN = '⠠'
```

The Braille character ⠠, used to indicate that the following letter should be capitalized.

```
BRaille_NUMBERS = {
    # Numbers
    '0': '⠼', '1': '⠠', '2': '⠠', '3': '⠠', '4': '⠠',
    '5': '⠠', '6': '⠠', '7': '⠠', '8': '⠠', '9': '⠠',
}
```

Dictionary mapping numeric digits (0-9) to their Braille equivalents.

```
NUMBER_SIGN = '⠼'
```

The sign ⠼ is the prefix for numeric sequences.

```
GRADE2_CONTRACTIONS = {
    'and': '⠠',
    'for': '⠠',
    'the': '⠠',
    'with': '⠠',
    'of': '⠠',
    'to': '⠠',
    'but': '⠠',
    'can': '⠠',
    'do': '⠠',
    'go': '⠠',
    'have': '⠠',
    'in': '⠠',
    'it': '⠠',
    'that': '⠠',
    'this': '⠠',
    'you': '⠠',
    'she': '⠠',
    'he': '⠠',
    'they': '⠠',
    'we': '⠠',
    # Add more contractions as needed
}
```

A dictionary defining common Grade 2 Braille word contractions, such as and, for, and the, which shorten frequent words for more efficient reading and writing.

```
def read_pos_files(grammar_dir):
    """
    Reads terminal words from separate POS files within the grammar directory.

    :param grammar_dir: Path to the directory containing POS .txt files.
    :return: Dictionary mapping non-terminals to lists of terminal words.
    """
    pos_files = {
        'Vi': 'Vi.txt',
        'Vt': 'Vt.txt',
        'DT': 'DT.txt',
        'NN': 'NN.txt',
        'PRP': 'PRP.txt',
        'PRE': 'PRE.txt',
        'Aux': 'Aux.txt',
        'Adv': 'Adv.txt',
        'CONJ': 'CONJ.txt',
        'NUM': 'NUM.txt',
        'NEG': 'NEG.txt',
        'Adj': 'Adj.txt',
    }

    lexicon = {}

    for non_terminal, filename in pos_files.items():
        file_path = os.path.join(grammar_dir, filename)
        try:
            with open(file_path, 'r', encoding='utf-8') as file:
                words = [line.strip() for line in file if line.strip()]
                quoted_words = [f"'{word}'" | '{word.capitalize()}'" for word in words]
                lexicon[non_terminal] = quoted_words
        except FileNotFoundError:
            print(f"Error: {filename} not found in {grammar_dir}.")
            exit(1)
        except Exception as e:
            print(f"Error reading {filename}: {e}")
            exit(1)

    return lexicon
```

Reads part-of-speech files from a given directory. Each file corresponds to a specific POS category (e.g., Vi.txt for intransitive verbs, NN.txt for nouns). Returns a dictionary mapping each POS category to a list of terminal words where each word is formatted with both its lowercase and capitalized form. In case of any file missing or error reading from a file, print an error message and stop the execution of the program. Then, return a populated lexicon dictionary.

```
def construct_grammar(lexicon):
    """
    Constructs the CFG grammar string by combining fixed rules with terminal words.

    :param lexicon: Dictionary mapping non-terminals to terminal words.
    :return: A string representing the CFG.
    """
    # Fixed grammar rules
    '''
    S=sentence, VP=verb phrase, NP=noun phrase,
    PP=prepositional phrase, DT=determiner, Vi=intransitive verb,
    Vt=transitive verb, NN=noun, PRP=Pronoun, CONJ=conjunction, Aux=auxiliary verb,
    Adv=adverb, Adj=adjective, PRE=Preposition
    NEG=negatio, NUM=number
    '''
    grammar_rules = """
    S -> NP VP
    S -> NP
    S -> VP
    S -> NUM
    S -> S CONJ S
    CONJ -> CONJ CONJ

    VP -> Vi
    VP -> Vt
    NP -> Vt NP
```

```

VP -> Vt Adv
VP -> Aux VP
VP -> Adj VP
VP -> Aux NEG VP
VP -> Vt PP
VP -> VP CONJ VP
VP -> VP NEG VP
VP -> PRE VP

NP -> DT NN
NP -> NP PP
NP -> NN
NP -> NUM NN
NP -> NP CONJ NP
NP -> Adj NN
NP -> DT Adj NN
NP -> NP Aux PRP
NP -> PRP NN

PP -> PRE NP
PP -> PRE VP

NUM -> NUM NUM
"""

# Add terminal productions from lexicon
for non_terminal, words in lexicon.items():
    # Join words with ' | ' to denote alternatives
    terminals = ' | '.join(words)
    grammar_rules += f"{non_terminal} -> {terminals}\n"

return grammar_rules

```

The `construct_grammar` function generates a Context-Free Grammar as a string by combining pre-defined grammar rules with terminal words from a given lexicon. It first defines some basic grammar rules for sentence structures and phrase components. Then it loops through the lexicon, adding terminal productions for each non-terminal, for example, Vi, Vt, DT, NN, using the words from the lexicon. This appends each set of words joined with `|` representing alternatives to the `grammar_rules` string. It then finally returns the whole CFG as a string.

```

def preprocess_sentence(sentence):
    """
    Preprocesses a sentence to ensure numbers are separated and properly tokenized.
    """
    sentence = re.sub(r'(\d+)', r' \1 ', sentence)
    return sentence

```

The function `preprocess_sentence` ensures that numbers in a sentence are properly separated and tokenized. Using a regular expression, it replaces any occurrence of one or more digits (`\d+`) by that number surrounded by spaces to make sure that numbers are treated as individual tokens. Therefore, the function returns the modified sentence with space around numbers.

```

def custom_tokenize(sentence):
    """
    Custom tokenizer that splits multi-digit numbers into individual digits
    and retains other tokens as-is.

    :param sentence: The input sentence to tokenize.
    :return: A list of tokens.
    """
    tokens = []
    for word in sentence.split():
        if word.isdigit(): # Check if the word is a multi-digit number
            tokens.extend(list(word)) # Split digits into individual tokens
        else:
            tokens.append(word) # Add non-digit words as-is
    return tokens

```

The `custom_tokenize` function splits a sentence into tokens. It keeps in mind the multi-digit numbers and breaks them down into individual digits. For each word in the sentence, it checks if it's a number using `isdigit()`. If it is, the number is broken down into its constituent digits, and each digit is added as a separate token. Non-numeric words are kept as-is. This will ensure that the multi-digit numbers are tokenized more granularly, ensuring the integrity of the other words remains intact.

```
def parse_sentence(parser, sentence):
    """
    Parses a sentence and prints all possible parse trees.

    :param parser: An NLTK parser object.
    :param sentence: A string representing the sentence to parse.
    :return: Parsed sentence as a string.
    """
    # Preprocess the sentence to handle numbers correctly
    sentence = preprocess_sentence(sentence)

    # tokens = nltk.word_tokenize(sentence)
    tokens = custom_tokenize(sentence) # Use custom tokenizer
    print(f"\nParsing sentence: '{sentence}'")
    print(f"\nTokenized sentence: {tokens}")
    parse_trees = list(parser.parse(tokens))

    try:
        parse_trees = list(parser.parse(tokens))
    except Exception as e:
        print(f"Error parsing sentence: {e}")
        return None

    if not parse_trees:
        print("No parse trees found.")
        return None
    else:
        for idx, tree in enumerate(parse_trees, 1):
            print(f"\nParse Tree {idx}:")
            print(tree)
            tree.draw()
        return sentence # Return the sentence for translation
```

This will parse a sentence with a given parser and print all possible parse trees. It will first preprocess the sentence to treat numbers properly: it makes sure that numbers get tokenized individually. Then it uses a customized tokenizer (`custom_tokenize`) that tokenizes this sentence. Afterwards, it does its best effort parsing using the given parser (`parser.parse`). If no parse trees are found, it prints a message to that effect. Then, if parse trees are generated, the function iterates through them, prints each one, and displays each graphically with `tree.draw()`. In the case of a parsing error, the function catches exceptions and prints the error message. The function returns the original sentence, likely for further processing or translation.

```
def translate_to_grade2_braille(text):
    """
    Translates English text to a simplified version of Grade 2 Braille.
    This implementation handles a subset of Grade 2 contractions and numeric content.

    :param text: The English text to translate.
    :return: A string representing the translated Braille.
    """
    braille = ''
    words = text.split()

    for word in words:
        word_braille = ''
        original_word = word # Save original for debugging

        # Handle numbers (Changed this as well (Galih))
        if word.isdigit():
            word_braille += NUMBER_SIGN # Add numeric mode indicator
            for digit in word:
```

```

        word_braille += BRAILLE_NUMBERS.get(digit, '')

    # Add space to separate numbers from subsequent letters
    braille += word_braille + ' ' # Ensure a space follows numbers
    continue

    # Check if the word is capitalized
    if word[0].isupper():
        word_braille += CAPITAL_SIGN
        word = word.lower()

    # Check if the entire word has a contraction
    if word in GRADE2_CONTRACTIONS:
        print(f"Word-level contraction found: '{original_word}' -> '{GRADE2_CONTRACTIONS[word]}'")
        word_braille += GRADE2_CONTRACTIONS[word]
    else:
        # Check for contractions within the word
        i = 0
        while i < len(word):
            contraction_found = False
            for contraction in sorted(GRADE2_CONTRACTIONS.keys(), key=lambda x: len(x), reverse=True):
                contraction_end = i + len(contraction)
                # Match contraction only if it fits completely and is bounded
                if (
                    word[i:contraction_end] == contraction
                    and (i == 0 or not word[i-1].isalpha()) # Start boundary check
                    and (contraction_end == len(word) or not word[contraction_end].isalpha()) # End boundary check
                ):
                    # Log contraction match
                    print(f"Contraction match: '{word[i:contraction_end]}' in '{original_word}' -> '{GRADE2_CONTRACTIONS[contraction]}'")
                    word_braille += GRADE2_CONTRACTIONS[contraction]
                    i += len(contraction)
                    contraction_found = True
                    break
            if not contraction_found:
                # Map individual letters
                char = word[i]
                braille_char = GRADE1_BRAILLE.get(char, '')
                if braille_char:
                    print(f"Letter match: '{char}' -> '{braille_char}'")
                else:
                    print(f"No Braille mapping for character: '{char}'")
                word_braille += braille_char
                i += 1

        braille += word_braille + ' '

    return braille.strip()

```

The `translate_to_grade2_braille` function converts the text from English into a simplified Grade 2 Braille. It processes the text one word at a time, including numbers and contractions. First, it checks if the word is numeric and adds a number sign (NUMBER_SIGN), converting each digit to its Braille equivalent. If the word is capitalized, it adds a capital sign (CAPITAL_SIGN) and converts the word to lowercase for further processing. It then searches for Grade 2 contractions within the word and, if found, it replaces the word with its equivalent Braille contraction. If a contraction is not found, then it cycles through the word character by character, changing each letter into its Grade 1 Braille equivalent. It makes certain that the contractions only match at the word boundaries and adds proper spacing between words and numbers. Finally, it returns the Braille translation of the input text.

```

def translate_from_braille(braille_text):
    """
    Translates Braille text back to English, handling Grade 2 contractions on a word basis.
    If a word doesn't match a Grade 2 contraction, it is translated letter-by-letter (Grade 1).

    :param braille_text: The Braille text to translate.
    :return: A string representing the English translation.
    """
    text = ''
    words = braille_text.split(' ') # Use space to split words

```

```

for word in words:
    word_text = ''
    i = 0 # Initialize character pointer
    capitalized = False # Flag to track capitalization

    while i < len(word):
        # Check for capital sign
        # Handle capital sign
        if word[i] == CAPITAL_SIGN:
            i += 1 # Move to the next character
            if word[i] in BRAILLE_TO_GRADE2:
                # Translate contraction after capital sign
                contraction_text = BRAILLE_TO_GRADE2[word[i]]
                word_text += contraction_text.capitalize() # Capitalize the contraction
                i += 1 # Skip the contraction character
                continue
            else:
                capitalized = True # Set capital flag for next individual character

        # Check for number sign
        if word[i] == NUMBER_SIGN:
            i += 1
            while i < len(word) and word[i] in BRAILLE_NUMBERS.values():
                for digit, braille_char in BRAILLE_NUMBERS.items():
                    if word[i] == braille_char:
                        word_text += digit
                        break
                i += 1
            continue

        # Handle Grade 2 contractions
        if word in BRAILLE_TO_GRADE2:
            print(f"Word-level contraction found: '{word}' -> '{BRAILLE_TO_GRADE2[word]}'")
            word_text = BRAILLE_TO_GRADE2[word]
            capitalized = False # Reset capitalization after applying
            break # Move to the next word

        # Decode individual letters (Grade 1 Braille)
        char_braille = word[i]
        char_text = BRAILLE_TO_GRADE1.get(char_braille, None)
        if char_text:
            # Apply capitalization if the flag is set
            if capitalized:
                char_text = char_text.upper()
                capitalized = False # Reset capitalization after applying
            word_text += char_text
        else:
            print(f"No English mapping for Braille character: '{char_braille}'")
            i += 1

        text += word_text + ' '

    return text.strip()

```

The `translate_from_braille` function will convert the Braille text back into English. It supports Grade 2 Braille contractions as well as individual Grade 1 Braille characters. It splits the input Braille by spaces to handle each word one by one. For every word, it iterates over every character in the Braille in search of the special symbols for capital sign (`CAPITAL_SIGN`) and number sign (`NUMBER_SIGN`). The capital sign at the beginning will indicate that the subsequent contraction should be written in capital. If it encounters a number sign, it translates Braille digits into their numeric equivalent. Then it checks if the whole word has a Grade 2 contraction translation and translates it that way. Otherwise, it proceeds to decode each character from Braille to English based on Grade 1 Braille mapping. It also provides for capitalizing individual letters when it encounters a capital sign. The result is a full English translation of the Braille text, with appropriate capitalization and numerical representation.

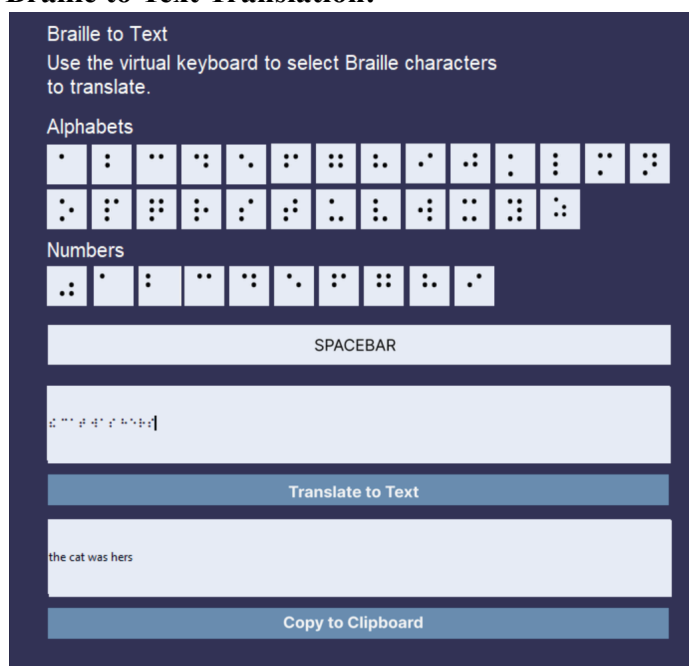
B. Datasets

The datasets used in our project comprise of the words/phrases lists used for each grammar role. For word classes like Adjective, Noun, and Adverb, the

list of words are extracted from a github repository that we have found (<https://github.com/taikuukaits/SimpleWordlists/tree/master>). However, other word classes do not have pre-built or publicly available datasets or text files containing lists of words. Therefore, we have manually scraped words online and transformed it into a text file for each class that is not found.

C. Experimental (Test Run) Analysis and Results

1. Braille to Text Translation:



Braille to Text

Use the virtual keyboard to select Braille characters to translate.

Alphabets

Numbers

SPACEBAR

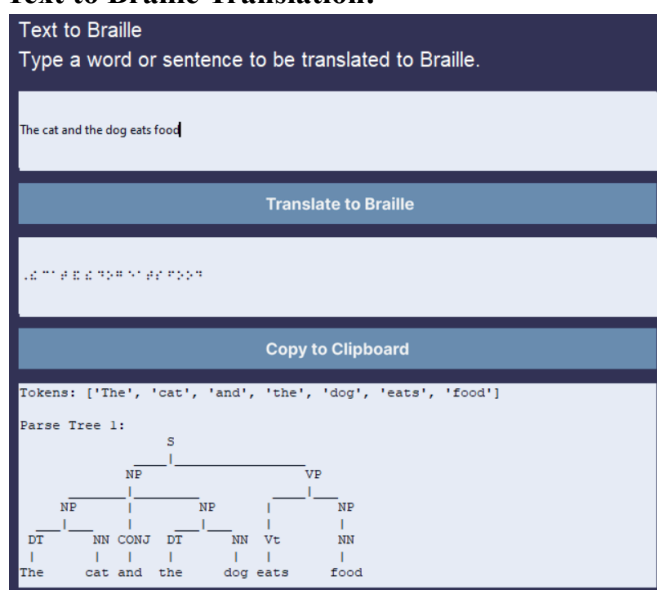
the cat was hers

Translate to Text

Copy to Clipboard

The function that translates braille to text is displayed above. The user inputs the desired braille sentence using the virtual keyboard, then presses on 'Translate to Text' to run the function. The function works as intended, where the output is displayed on the box below. It can be copied to the user's clipboard using the button below it.

2. Text to Braille Translation:



Text to Braille

Type a word or sentence to be translated to Braille.

The cat and the dog eats food

Translate to Braille

Copy to Clipboard

Tokens: ['The', 'cat', 'and', 'the', 'dog', 'eats', 'food']

Parse Tree 1:

```

graph TD
    S --> NP1[NP]
    S --> VP[VP]
    NP1 --> DT1[DT]
    NP1 --> NN1[NN]
    NP1 --> CONJ[CONJ]
    NP1 --> NP2[NP]
    NP2 --> DT2[DT]
    NP2 --> NN2[NN]
    VP --> Vt[Vt]
    VP --> NP3[NP]
    NP3 --> NN3[NN]
    DT1 --> The[The]
    NN1 --> cat[cat]
    CONJ --> and[and]
    DT2 --> the[the]
    NN2 --> dog[dog]
    Vt --> eats[eats]
    NN3 --> food[food]
  
```

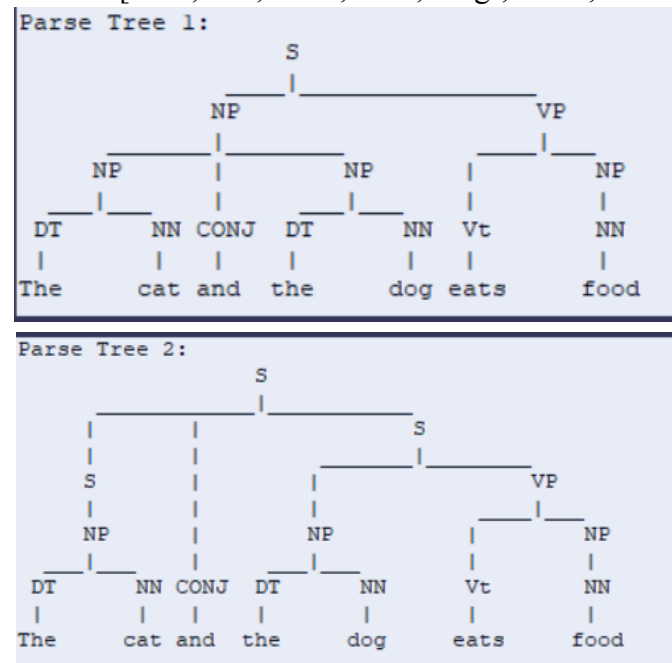

The function that translates text to braille is displayed above. The user inputs the desired text sentence, then presses on ‘Translate to Braille’ to run the function. The function works as intended, where the output is displayed on the box below. The resulting parse tree(s) are also displayed.

3. Parse Tree Outputs:

Each node of the tree represents the grammar construct, letters like sentences (S), noun phrase (NP), verb phrase (VP), and etc same with the given CFGs from the previous sections.

Parse Tree Output: “The cat eats food”

Tokens: ['The', 'cat', 'and', 'the', 'dog', 'eats', 'food']



The figures above are the Parse trees for the input of “The cat eats food”. Due to the grammar’s ambiguity, there can be more than one tree created.

V. Discussions

A. Implications

The results demonstrate that our code was successful in implementing text-to-braille and braille-to-text translation for users, using computational techniques like lexer and parser, and creating a context-free grammar that adheres to the rules of the English Language.

B. Limitations

Although our program functions as intended, the English Language is made up of countless grammar structures that make it up. This results in our grammar not being able to create complex inputs that consist of multiple sentences which may result in more complex grammar structures that do not exist yet in our CFG. Additionally, the constantly expanding amount of words used in the English Language is beyond the scope of the mere several thousand words

used in our dataset for tokenization. Our program is only designed for single sentences at a time.

Punctuation is also unusable in our program, due to the complexity of its nature in the English language. As of the report submission, we are unable to integrate the usage of punctuation yet.

VI. Conclusion and Recommendation

The project created, Braille Translator, is basically a desktop application designed to translate the English text to Braille (Level 2) and vice versa. It aims at creating awareness and improving the inclusivity for the visually impaired. The translation of English sentences into Braille and generation of parse trees in the application relies on computational techniques followed by lexical and syntactic analyses using a lexer and a parser, respectively. Also, the app helps users learn Braille basics with its user-friendly interface.

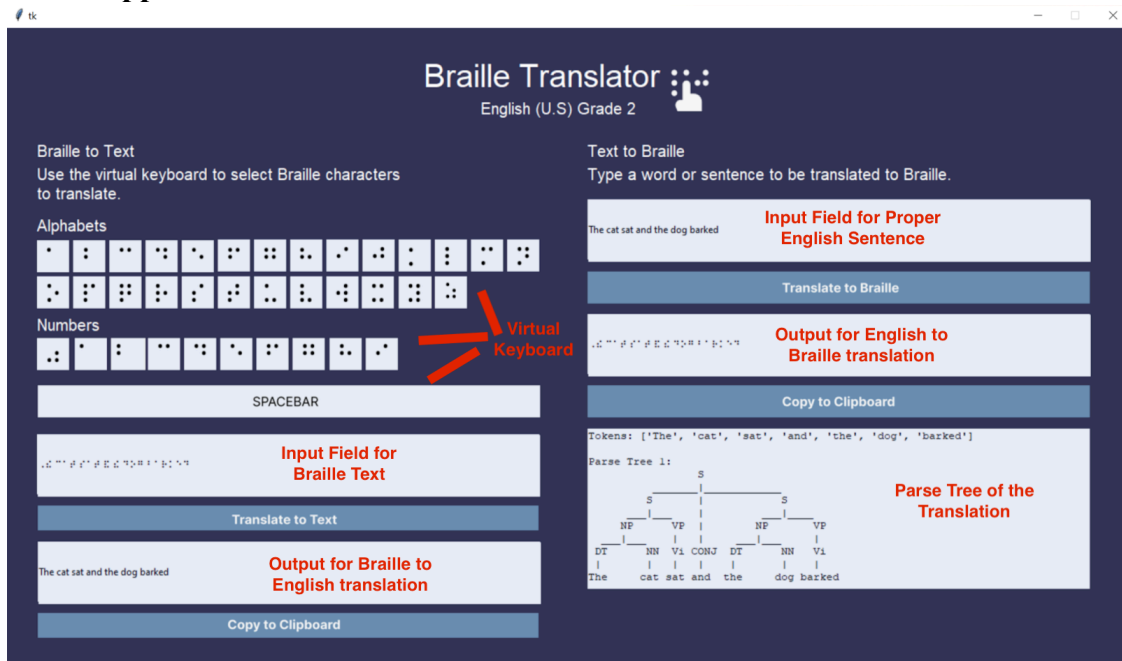
The system fills an important gap by providing a platform supporting learning and translation to Braille, in order to bring people closer to understanding the complexity and importance of Braille literacy. With this application, one can know the basics of how a blind or visually impaired person reads and writes and develop much-needed empathy and awareness within society at large.

However, it is not without its limitations, such as the absence of punctuation and being limited to single sentences. These could be improved in order to further increase the usability and scalability of our program.

In later stages of this English-to-Braille project, it is recommended that the grammatical structure be expanded to accommodate more complex grammatical structures in the English language, and to allow multiple sentences. In addition to that, more and more words can be added to the various .txt grammar role files to increase scalability.

VII. Program Manual (Screenshots)

A. Application Interface



1. Virtual Keyboard (Braille Input): Placed on the left side of the screen and allows users to input Braille characters by clicking on the virtual keyboard. It has alphabet and numeric characters available.
2. Input Fields:
 - a. Braille to Text Input Field (Left): This is where users can input Braille text to be translated to English.
 - b. Text to Braille Input Field (Right): This is where users can type a proper English sentence to be converted to Braille.
3. Translation Buttons:
 - a. Translate to Text (Left): Translates Braille input to English.
 - b. Translate to Braille (Right): Translates English input to Braille.
4. Output Areas:
 - a. Output for Braille to English Translation (Left): Displays the English text equivalent to the Braille input.
 - b. Output for English to Braille Translation (Right): Displays the Braille equivalent of the English sentence.
5. Parse Tree Section: This section displays the grammatical structure of the English sentence through a parse tree.
6. Copy to Clipboard Functionality: Both outputs have a "Copy to Clipboard" button so the translated text can be easily copied for further use.

B. How to Use the Braille Translator Application

Step 1: Download or Clone the Repository

1. Download

- Go to the Github repository (link).
- Click on the green 'Code' button and select 'Download ZIP'.
- Extract the ZIP file to your desired location on your device.

OR

2. Clone

- Open your terminal or command prompt.

- Run the following command to clone the repository:

```
git clone
https://github.com/rafaelsutiono/braille-t
ranslator.git
```

Step 2: Change File Directory in Code

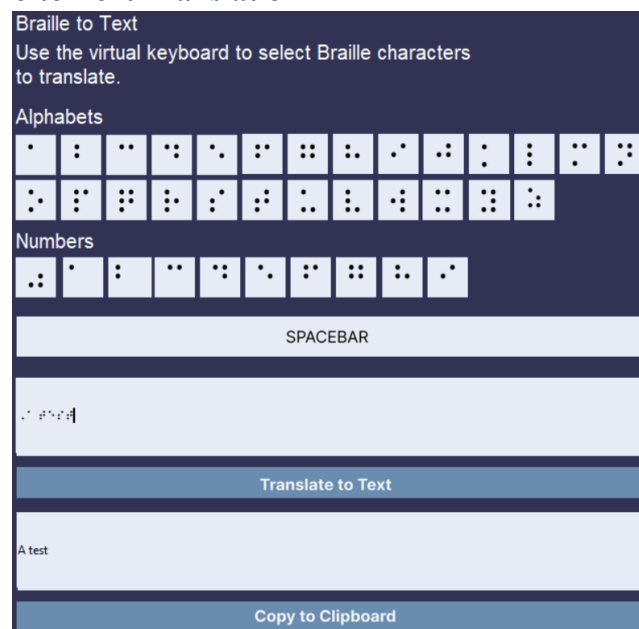
- In the gui.py file (build folder), change the code on line 26 to the file directory that leads to the file build/assets/frame0 in your laptop.
- The line begins with ASSETS_PATH =

```
26 ASSETS_PATH = OUTPUT_PATH / Path(r"C:\JohnDoe\braille-translator\build\assets\frame0") # change this to the correct frame0 path on your machine
27
```

Step 3: Run the gui.py file

- Run the simulator with the following command:
 - `python build/gui.py`

Step 4a: Braille-to-Text Translation



- Using the virtual keyboard on the top half of the screen, type in your desired input sentence.
- Click on 'Translate to Text'.
- The output will be displayed on the box at the bottom. Click on 'Copy to Clipboard' to copy the output to your clipboard.

Step 4b: Text-to-Braille Translation

Text to Braille
Type a word or sentence to be translated to Braille.

A test

Translate to Braille

.A test

Copy to Clipboard

Tokens: ['A', 'test']

Parse Tree 1:

```

      S
      |
      NP
      |
  DT  |  NN
  |   |   |
  A   test

```

- Click on the box on the very top so that your cursor appears on it.
- Type your sentence in proper English Grammar.
- When finished, click on 'Translate to Braille'
- The braille output is displayed on the box in the middle. Click on 'Copy to Clipboard' to copy it to your clipboard.
- The resulting parse tree(s) are displayed on the box at the bottom.
- If a word doesn't exist, the following is displayed and the output will be empty:

Error: Grammar does not cover some of the input words: "'wghstrfg'".

- If a grammar structure doesn't exist, the following is displayed and the output will be empty:

No valid parse trees found.

VIII. External Links:

- Github Project Repository : <https://github.com/rafaelsutiono/braille-translator>
- Application Demo Video : [YouTube](#) Compilation Technique Final Project Demo
- Final Presentation: [Canva](#)

IX. References

- GeeksforGeeks. (2024, January 3). *Lexical analyser in C*. GeeksforGeeks. <https://www.geeksforgeeks.org/c-lexical-analyser-lexer/>
- GeeksforGeeks. (2024b, October 24). *Parsing | Set 1 (Introduction, Ambiguity and Parsers)*. GeeksforGeeks. <https://www.geeksforgeeks.org/introduction-of-parsing-ambiguity-and-parsers-set-1/>
- Jiménez, J., Olea, J., Torres, J., Alonso, I., Harder, D., & Fischer, K. (2009). Biography of Louis Braille and Invention of the Braille Alphabet. *Survey of Ophthalmology*, 54(1), 142–149. <https://doi.org/10.1016/j.survophthal.2008.10.006>
- Nahar, L., Jaafar, A., & Sulaiman, R. (2019). USABILITY EVALUATION OF a MOBILE PHONE BASED BRAILLE LEARNING APPLICATION “MBRAILLE.” *Malaysian Journal of Computer Science*, 108–117. <https://doi.org/10.22452/mjcs.sp2019no1.8>
- Nahar, L., Jaafar, A., Ahamed, E., & Kaish, A. B. M. A. (2015). Design of a Braille learning application for visually impaired students in Bangladesh. *Assistive Technology*, 27(3), 172–182. <https://doi.org/10.1080/10400435.2015.1011758>