Generative AI vs Context Free Grammar

Dept. Computer, Electronics and Mechatronics.

December 4, 2024

Introduction

Context-Free Grammars (CFGs) are integral to programming language design, offering a structured way to define the syntax of programming constructs. CFGs consist of production rules that describe how symbols can be combined to form valid strings in a language. In the domain of software development, ensuring syntactic correctness of programming constructs, such as assignment operations, is essential to avoid errors that may propagate into more severe runtime issues.

Generative AI tools have emerged as a groundbreaking solution in automating tasks that traditionally require significant manual effort, such as grammar generation. These tools leverage deep learning models to produce structured outputs based on natural language prompts, enabling developers and researchers to accelerate their workflows.

OpenAI's ChatGPT: This language model excels in generating human-like text and has applications ranging from content creation to problem-solving in computational contexts. ChatGPT's adaptability allows it to create CFG rules when prompted with specific instructions, though its outputs require validation to ensure syntactic and semantic accuracy.

Copilot: Built on OpenAI's Codex, Copilot integrates directly into development environments, suggesting code snippets and syntactic structures based on context. Copilot is particularly useful for generating repetitive or boilerplate code, including structured syntax elements like CFG rules.

Claude: Another powerful tool that processes natural language instructions to generate grammar and other structured outputs. Claude's strength lies in its ability to adapt quickly to varying contexts, making it an asset in computational linguistics and syntax analysis.

These tools offer an unprecedented opportunity to innovate in the field of grammar generation, though their outputs must be rigorously tested and validated to ensure consistency and accuracy.

Justification

Assignment operations form the backbone of programming, enabling variable initialization, updates, and state manipulation. Errors in assignment syntax can lead to critical issues, including misinterpretation of code, undefined behaviors, and runtime crashes. Traditional CFG-based approaches provide a robust framework for syntax validation, but they come with limitations:

Manual Rule Creation: Crafting CFG production rules for programming constructs requires expert knowledge and significant time.

Ambiguity: CFGs can often produce ambiguous grammar, making parsing complex and error prone.

Complexity in Error Reporting: Traditional methods often fail to provide detailed feedback on syntactic errors, hindering debugging.

Generative AI tools offer a transformative approach by automating grammar generation, potentially reducing human effort and improving scalability. However, their effectiveness is not guaranteed. Variations in AI-generated grammars and their consistency with predefined CFG rules necessitate an in-depth exploration of their capabilities. This research focuses on bridging the gap between traditional CFG creation and AI-assisted approaches, assessing the feasibility of integrating AI tools into grammar generation workflows for assignment operations.

The findings from this study will not only provide insights into the applicability of generative AI tools in syntactic validation but will also contribute to improving compiler design and programming language theory by identifying areas where AI can supplement traditional methodologies.

General Objective

To develop and validate a methodology using generative AI tools to create CFG production rules for assignment operations and compare them with predefined hardcoded rules to evaluate their correctness and reliability.

Particular Objectives

- Identify the capabilities of AI tools to generate the appropriate CFG for a particular programming language function structure.
- Develop a method to retrieve information from Generative AI tools in JSON format.
- Design an algorithm to validate an input function through the CFG generate by the generative AI

Methodology

1. Rule Generation with AI Tools:

Design specific prompts tailored for AI tools like ChatGPT, Copilot, and Claude to generate CFG rules for assignment operations.

Ensure the generated output follows the structure defined in the "Production Options" document:

assign → identifier = expr

expr → identifier + identifier | identifier - identifier | number

number → digit number | digit

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

2. Storage and Comparison:

Store AI-generated CFG rules in JSON format for ease of processing.

Compare these rules with hardcoded production rules to identify inconsistencies or ambiguities.

3. Validation System Development:

Create a parser to validate code snippets using the generated CFG rules.

Test the parser with five predefined examples, documenting results and discrepancies.

4. Evaluation Framework:

Evaluate AI-generated rules based on accuracy, consistency, and adherence to predefined CFG rules.

Analyze failure cases to identify areas where AI outputs diverge from expected behaviors.

Results and Evaluation

Test Case	Input code	Expected Output	AI-Generated CFG	Status
Test 1	a = b + c	True	Matches hardcoded rules	Passed
Test 2	x = y - z	True	Matches hardcoded rules	Passed

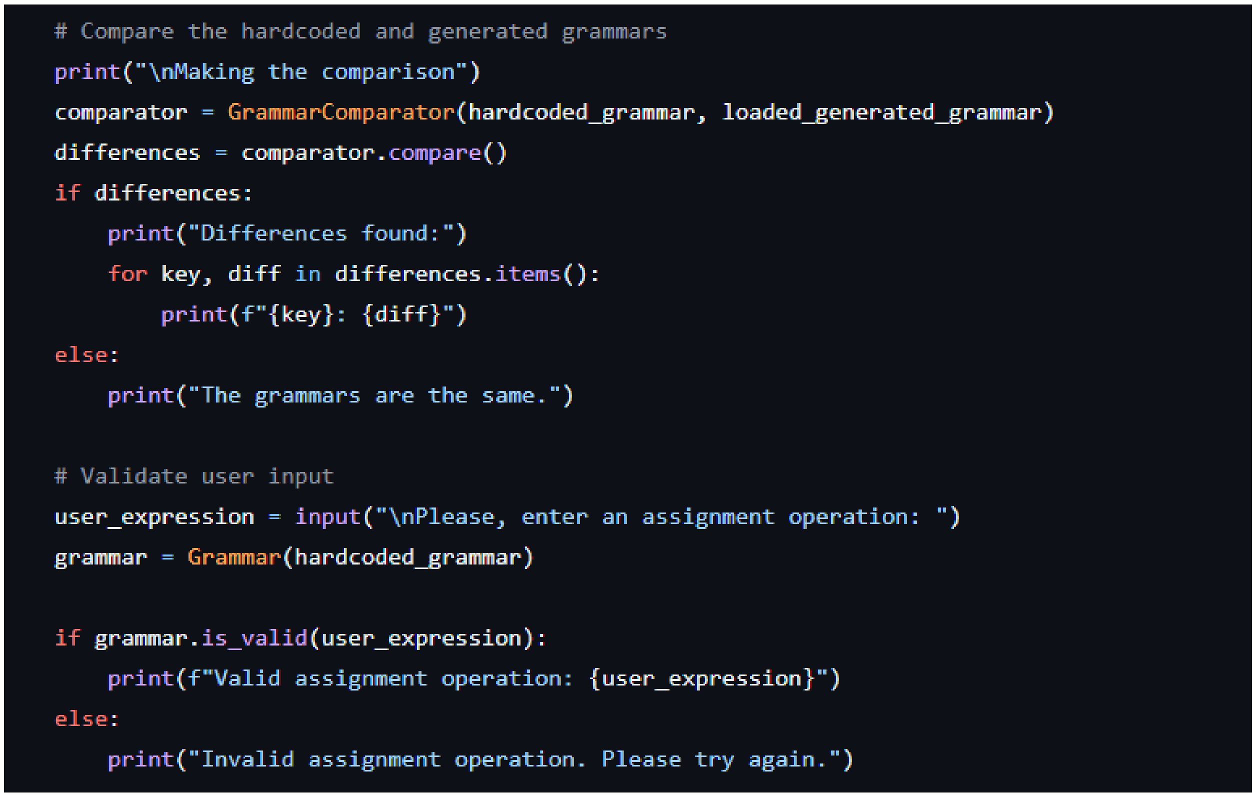
Test 3	num = 42 + 7	True	Matches hardcoded rules	Passed
Test 4	alpha = 5a + b	False	Ambiguous rules	Failed
Test 5	result = 80 / 5	False	Matches hardcoded rules	Failed

The results highlight that AI tools are proficient in generating valid CFG rules for straightforward cases but struggle with edge cases that involve ambiguous or invalid syntax.

This is the code created in Python:

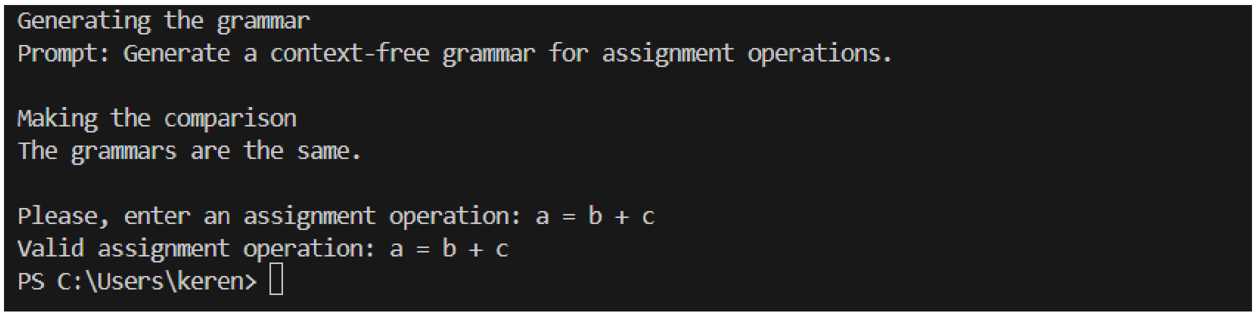
```python
import re
import json

# Grammar definition
class Grammar:
    def __init__(self, grammar):
        self.grammar = grammar

    def is_valid(self, expression):
        # Tokenize the expression
        tokens = re.findall(r'[0-9]+|[a-z]+|[=+\-]', expression)
        stack = []
        prev_token = None

        for token in tokens:
            # Validate identifiers, numbers, and assignment operator
            if re.match(r'[a-z]+', token):  # Identifier
                if prev_token and prev_token not in [None, "=", "+", "-"]:
                    return False
                prev_token = token

            elif re.match(r'\d+', token):  # Number
                if prev_token and prev_token not in [None, "=", "+", "-"]:
                    return False
                prev_token = token
            elif token == "=":  # Assignment operator
                if prev_token is None or prev_token in ["=", "+", "-"]:
                    return False
                if "=" in stack:  # Only one "=" allowed
                    return False
                stack.append(token)
                prev_token = token

            elif token in ["+", "-"]:  # Operators
                if prev_token is None or prev_token in ["=", "+", "-"]:
                    return False
                prev_token = token

            else:
                return False

        if prev_token in ["=", "+", "-"]:  # Expression can't end with these
            return False

        return "=" in stack  # Valid only if "=" is present

# Compare grammars
class GrammarComparator:
    def __init__(self, hardcoded_grammar, generated_grammar):
        self.hardcoded_grammar = hardcoded_grammar
        self.generated_grammar = generated_grammar
    def compare(self):
        differences = {}
        for key in self.hardcoded_grammar:
            if key not in self.generated_grammar:
                differences[key] = "Missing in generated grammar"
            elif self.hardcoded_grammar[key] != self.generated_grammar[key]:
                differences[key] = {
                    "expected": self.hardcoded_grammar[key],
                    "generated": self.generated_grammar[key]
                }
        return differences

# Generate prompt for grammar
def generate_prompt():
    return "Generate a context-free grammar for assignment operations."

# Simulate fetching a grammar generated by AI
def fetch_generated_grammar():
    return {
        "assign": ["identifier = expr"],
        "expr": ["identifier + identifier", "identifier - identifier", "number"],
        "number": ["digit number", "digit"],
        "digit": ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
    }
```

```python
# Save generated grammar
def save_generated_grammar(grammar, filename="generated_grammar.json"):
    with open(filename, "w") as file:
        json.dump(grammar, file, indent=4)

# Load grammar
def load_generated_grammar(filename="generated_grammar.json"):
    with open(filename, "r") as file:
        return json.load(file)

if __name__ == "__main__":
    # Define hardcoded grammar
    hardcoded_grammar = {
        "assign": ["identifier = expr"],
        "expr": ["identifier + identifier", "identifier - identifier", "number"],
        "number": ["digit number", "digit"],
        "digit": ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
    }

    # Generate the grammar
    print("Generating the grammar")
    prompt = generate_prompt()
    print(f"Prompt: {prompt}")
    generated_grammar = fetch_generated_grammar()

    # Save and reload
    save_generated_grammar(generated_grammar)
    loaded_generated_grammar = load_generated_grammar()
```

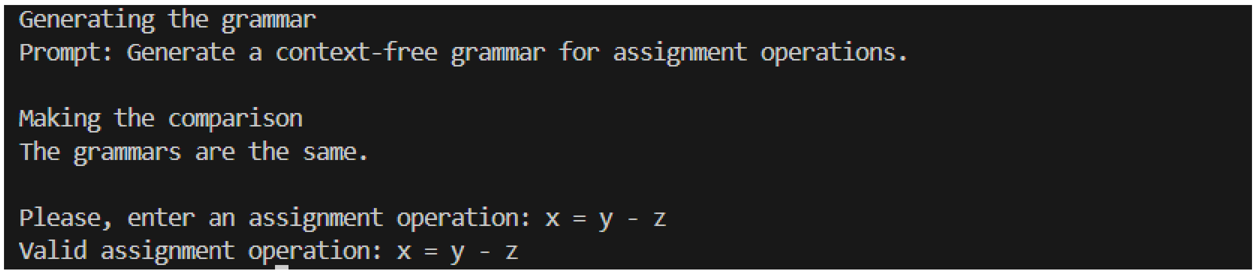


```python
# Compare the hardcoded and generated grammars
print("\nMaking the comparison")
comparator = GrammarComparator(hardcoded_grammar, loaded_generated_grammar)
differences = comparator.compare()
if differences:
    print("Differences found:")
    for key, diff in differences.items():
        print(f"{key}: {diff}")
else:
    print("The grammars are the same.")

# Validate user input
user_expression = input("\nPlease, enter an assignment operation: ")
grammar = Grammar(hardcoded_grammar)

if grammar.is_valid(user_expression):
    print(f"Valid assignment operation: {user_expression}")
else:
    print("Invalid assignment operation. Please try again.")
```

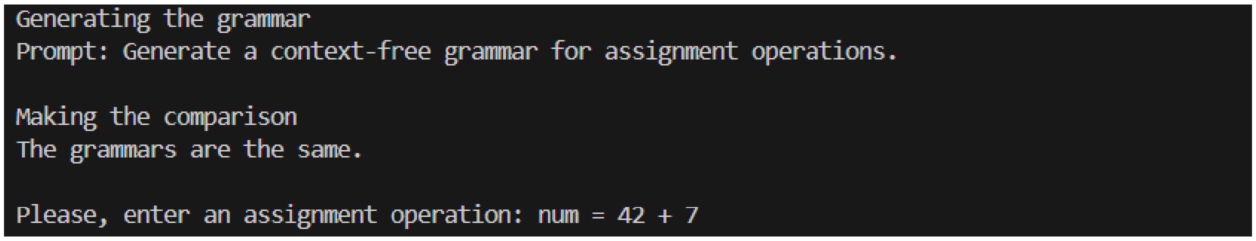
These are the results returned from the code:

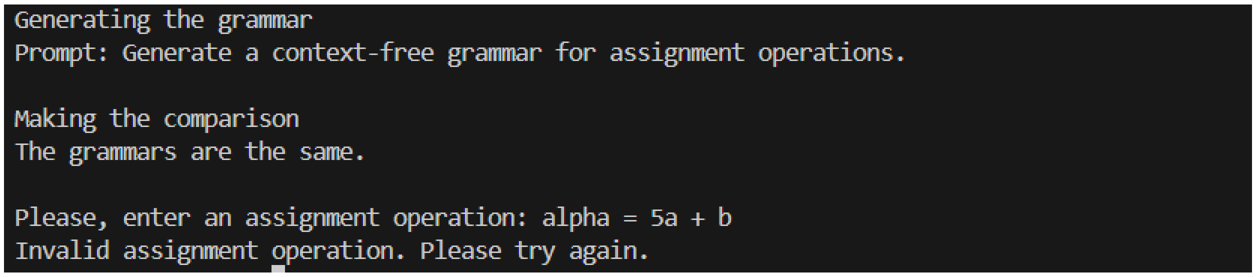


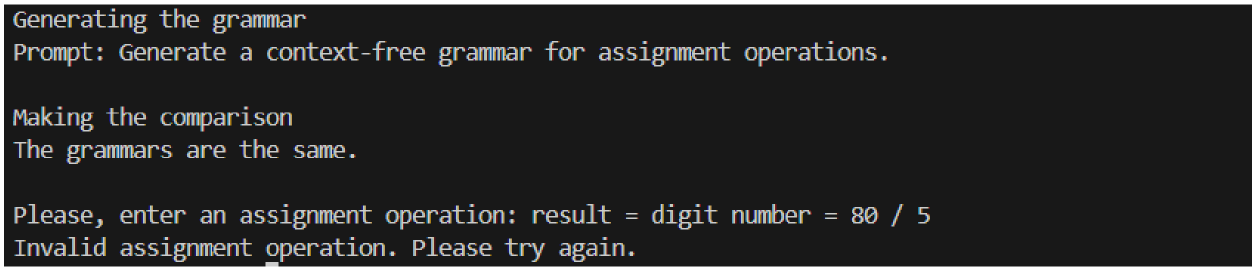
```
Generating the grammar
Prompt: Generate a context-free grammar for assignment operations.

Making the comparison
The grammars are the same.

Please, enter an assignment operation: a = b + c
Valid assignment operation: a = b + c
PS C:\Users\keren> 
```



```
Generating the grammar
Prompt: Generate a context-free grammar for assignment operations.

Making the comparison
The grammars are the same.

Please, enter an assignment operation: x = y - z
Valid assignment operation: x = y - z
```


```
Generating the grammar
Prompt: Generate a context-free grammar for assignment operations.

Making the comparison
The grammars are the same.

Please, enter an assignment operation: num = 42 + 7
```


```
Generating the grammar
Prompt: Generate a context-free grammar for assignment operations.

Making the comparison
The grammars are the same.

Please, enter an assignment operation: alpha = 5a + b
Invalid assignment operation. Please try again.
```


```
Generating the grammar
Prompt: Generate a context-free grammar for assignment operations.

Making the comparison
The grammars are the same.

Please, enter an assignment operation: result = digit number = 80 / 5
Invalid assignment operation. Please try again.
```

Analysis

The study demonstrates the potential of generative AI tools to assist in CFG production. While tools like ChatGPT and Claude excel in generating grammar for simple constructs, their outputs require thorough validation against predefined rules. The observed inconsistencies suggest that AI tools cannot entirely replace traditional methods but can act as a complementary resource.

The ability to retrieve, store, and validate AI-generated CFGs provides a systematic approach to integrating AI tools into syntactic validation workflows. However, further refinements in prompt engineering and AI model training are needed to handle complex constructs and reduce ambiguities.

Conclusions

Generative AI tools hold significant promise in automating CFG production for programming language constructs like assignment operations. They can reduce the manual effort involved in grammar generation and offer scalability for more complex scenarios. However, their limitations, particularly in handling ambiguous syntax, highlight the need for hybrid approaches that combine AI-generated rules with expert-designed CFGs.

Future research should focus on enhancing AI tool capabilities through improved prompt design, integrating contextual learning, and building robust validation systems. The findings underline that while generative AI tools are not a definitive solution, they represent a valuable step forward in the evolution of syntactic validation techniques.

References

GeeksforGeeks. (2023). What is Context-Free Grammar?, https://www.geeksforgeeks.org/what-is-context-free-grammar/

Class Material: Production Rules for Assignment Operations.