Theory of Computation (CSC720)
Class Project on

PDA-Based HTML/XML Validator

A validator that checks if HTML/XML code is well-formed

Submitted by:

Nabin Bhandari

nabin.bhandari@trojans.dsu.edu

Student ID: 101213073

Submitted To:

Yong Wang

Professor/Associate Dean Beacom Graduate Programs

Coordinator MS Cyber Defense

Abstract

This project “PDA-Based HTML/XML Validator” presents the design and implementation of a PDA-based HTML/XML validator using Python programming language. The primary goal of this project is to design and demonstrate how core concepts of Theory of Computation, specifically, Context Free Languages (CFLs) and Pushdown Automata (PDA) can be applied in real world scenario to solve practical problems in structured data validation. Modern document types like HTML and XML use and follow a nested, hierarchical structure, making them a natural fit for context-free grammar. This validator also simulates a deterministic PDA that uses a stack to match opening and closing tags, which correctly handles self-closing tags and ignoring tag attributes. This project supports both raw string input and file-based validation through a command-line interface. This project has support to verify for mismatched, unexpected, or unclosed tags, allowing users to quickly identify structural issues. By connecting theoretical PDA behavior with real-world document parsing, we can say that this project offers lightweight validation utility.

Keywords: Context Free Languages (CFLs) and Pushdown Automata (PDA), Syntax Validation

Acknowledgment

I would like to express my sincere gratitude to our Professor **Yong Wang**, Associate Dean of Beacom Graduate Programs and Coordinator of MS Cyber Defense, for his guidance and continuous support throughout his classes. As the instructor of the Theory of Computation course, Professor Wang provided a strong foundation in automata and languages, Complexity Theory and Computability Theory, which were essential in understanding and applying the concepts used in this project titled “PDA-Based HTML/XML Validator”. This project assignment not only reinforced the theoretical principles taught in class but also allowed me to explore their real-world applications through implementation. I am truly thankful for his valuable teaching and mentorship.

Table of Contents

Table of Contents ---- 4
Table of Figures ---- 6
List of Table ---- 6
CHAPTER 1 – INTRODUCTION ---- 7
1.1 Introduction ---- 7
1.1.1 Real-life Examples ---- 7
1.1.2 Report includes ---- 7
1.2 Problem Statement ---- 7
1.2 Challenges Faced ---- 8
1.3 Objective ---- 9
CHAPTER 2 – BACKGROUND STUDY ---- 10
2.1 Background Study ---- 10
2.1.1 Pushdown Automaton (PDA) ---- 10
2.1.2 Syntax Validator ---- 11
CHAPTER 3 – PROJECT DESIGN ---- 12
3.1 Project ---- 12
3.2 Theoretical Basis ---- 12
3.3 Approach ---- 12
3.4 Features Implemented ---- 13
3.5 Working Mechanism ---- 14
3.6 Language and Tools ---- 15
3.7 Execution Instructions ---- 15
3.9 Project Example ---- 17
CHAPTER 4 – TESTING ---- 18
4.1 Results and Testing ---- 18
4.1.1 Test Case 1 (Raw Input) ---- 18
4.1.2 Test Case 2 (File Based) ---- 19
4.2 Outcomes ---- 21

CHAPTER 5 – CONCLUSION AND FUTURE RECOMMENDATION ---------- 22
5.1 Conclusion ---------- 22
5.2 Future recommendations ---------- 22
APPENDICES ---------- 23
Snippets of major source components ---------- 23

Table of Figures

Figure 1: Pushdown Automata with data stack

Figure 2: HTML file with correct structure

Figure 3: Output for first test case

Figure 4: XML file with incorrect structure

Figure 5: Output for second test case

List of Table

Table 1: Test Cases

CHAPTER 1 – INTRODUCTION

1.1 Introduction

This project titled "**PDA-Based HTML/XML Validator**" is an application of formal language theory in computer science. The main goal is to simulate Pushdown Automaton (PDA), how this can be used to recognize and validate structured languages such as markup languages like HTML and XML. As we know, these languages are based on nested tags and hierarchical structure, which are characteristics of context-free languages. This report presents the theoretical foundation and practical implementation of the project.

1.1.1 Real-life Examples

Lightweight web development tools for basic HTML validation.

XML-based configuration checking (e.g., Android/IOS layouts, RSS feeds).

1.1.2 Report includes

- An explanation of the PDA concept
- Relevance to Theory of Computation
- Implementation details in Python programming language
- Step-by-step walkthrough with examples
- Challenges, results, and future work

1.2 Problem Statement

Markup languages like HTML and XML are widely used in real-world applications and rely on a well-defined, hierarchical nested structure of tags. These tags should be properly opened, closed, and nested. This is essential for correct parsing and rendering by browsers, compilers, or any XML-based application. However, many lightweight tools lack simple, theory-driven mechanisms for verifying the structural validity of such files.

This project tries to cover and address the problem of validating the well-formedness syntax of HTML/XML by simulating a Pushdown Automaton (PDA) which is a computational model capable of recognizing context-free languages. The goal is to bridge the gap between theoretical computation models and real-world applications by implementing a PDA-based validator in Python programming language that accepts properly nested tag structures and rejects improperly written tag ones. The solution aims to provide learning the concepts of Theory of Computation, while also demonstrating a practical use of stack-based automata in modern document validation.

1.2 Challenges Faced

During this project I encountered different challenges. Some of the major challenges are described below:

1. **Accurately Handling Self-Closing Tags.**
   One of the main challenges doing this project was correctly identifying and processing self-closing tags like <img />,
, and <input />. These tags must be explicitly excluded from the stack operations to prevent false mismatch errors in our code.

2. **Filtering out irrelevant Content.**
   Another challenge is filtering out irrelevant content and attributes. Markup languages like HTML/XML often includes attributes within tags and in many cases, they may contain comments (like <?xml?> in XML file or <!DOCTYPE> in HTML file), and many other non-structural elements. Our machine has to ensure these were ignored or excluded from validation, that are required to carefully use of regular expressions and content filtering.

3. **Providing clear and precise feedback.**
   Instead of providing generic error message like "invalid structure" errors or any other type of error, the goal was to give precise feedback like in simple way so that user can easily understand, such as the exact tag index where a mismatch occurs or identifying unclosed elements. This required tracking tag positions and maintaining a meaningful error message format in our system.

4. **Handling malformed inputs gracefully.**

   To Deal with incomplete or malformed input like <div (missing closing bracket), <div . (user put period (.) instead of closing bracket), or random words without bracket like div, span, img or stray angle brackets posed edge-case challenges in our system. Our machine had to reject such inputs initially while avoiding crashes.

5. **Case-Insensitive Matching.**

   Since HTML is case-insensitive for tag names, the system needed to normalize tag comparisons (e.g., <DIV> to).

6. **Simulating PDA Behavior.**

   Translating theoretical PDA behavior in a Practical Environment or say into working Python logic, especially under real-world HTML irregularities. Process required careful balance between theoretical correctness and practical implementation.

1.3 Objective

The primary goal of this project is to demonstrate how a Pushdown Automaton (PDA) can be used to validate real-world structured data, specifically HTML and XML. This project is highly relevant to the fundamental concepts of Theory of Computation. This helps to explore the application of context-free grammar (CFLs) and PDA theory in practice way.

Markup Language like HTML/XML structures exhibit the properties of context-free languages, where proper nesting and matching of tags can be done and represented by a PDA's stack operations. Through this project, the concepts of PDA transitions, stack-based decision-making, and language recognition are not just theorized, but implemented in a way that closely mimics real parsers used in compilers and browsers.

CHAPTER 2 – BACKGROUND STUDY

2.1 Background Study

The fundamental theories, general concepts, and terminologies related to the project are discussed below:

2.1.1 Pushdown Automaton (PDA)

A Pushdown Automaton (PDA) is a computational model which is used to recognize context-free languages. It extends a finite automaton with a stack implementation, which allows it to handle nested structures such as parentheses or HTML/XML based tags. In this project, the PDA is simulated in Python programming language to track opening and closing tags using stack operations.

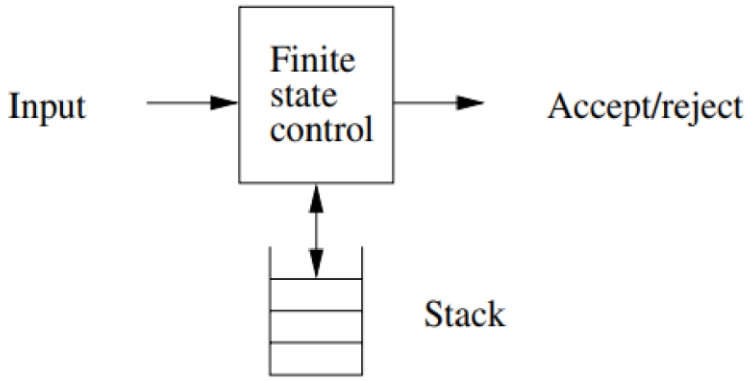

Figure 1: Pushdown Automata with data stack

From the figure above, we can say that PDA is a type of automata that has a stack as its memory. These stacks can be used to store information. This information is all about the position in the input string, information about the results of previous operations, or other information needed to process the input string given to the machine. In simple words, a PDA can be described as a tuple consisting of a state set, an input alphabet, a stack alphabet, a transition function, an initial state, an initial stack symbol, and a final state set. The pushdown automaton transition function maps different states, their input symbols, and stack symbols to the next state and maps a new sequence

of stack symbols. Stack operations like push and pop allow the machine to perform stack manipulations during its processing. The PDA can accept context-free languages, which can be generated by context-free grammar. PDA can accept context-free language. Here, context-free language is a language that can be generated by context-free grammar.

2.1.2 Syntax Validator

A syntax validator is a term used in this project as a tool that checks whether a given input follows the rules of a specific grammar or language structure. In this project, the validator ensures that HTML/XML based syntax when inputs have properly matched and verify the nested tags, treating the structure as a context-free language validated by PDA logic.

CHAPTER 3 – PROJECT DESIGN

3.1 Project

Design-Based Project using Programming Language (Python)

3.2 Theoretical Basis

The language of well-formed HTML/XML is a Context-Free Language (CFL). Context-Free Language are recognized by Pushdown Automata (PDAs), which use a stack to match nested structures. This project PDA-Based HTML/XML Validator simulates a deterministic PDA (DPDA) for validating markup languages like HTML/XML.

3.3 Approach

1. **Tokenize HTML/XML input to extract tags.**
   The first user gives input, this could be HTML or XML, is first tokenized and then extracted like tags.

2. **Use a stack to match opening and closing tags.**
   In this project stack is implemented to match opening and closing tags.

3. **Support self-closing tags and attributes.**
   This project supports self-closing tags and attributes.

4. **Provide both direct input and file-based validation.**
   This project can take input directly from user in terminal or can use file to validate the correct structure in a file.

5. **Show diagnostics for mismatched or unclosed tags.**
   This machine can show where the error came from.

3.4 Features Implemented

1. PDA simulation using stack operations

- Implemented pushdown automaton to validate nested tag structures using a stack.
- In stack operation opening tags are pushed, closing tags are popped, and self-closing tags are ignored by this system.

2. Command-Line interface with looped input

- This helps the user to validate their code continuously, with no need to recompile the project for a second time.
- Provides an interactive terminal interface where users can enter commands in a loop.

3. Supports file and raw HTML/XML input

- Users can input raw HTML/XML markup directly or can be able to validate content from a file using validate <filename>.

4. Mismatch detection with custom error messages

Detects and reports:

- Unexpected closing tags error
- Incorrectly matched pairs error
- Unclosed tags remaining on the stack error
- Use of closing tags on self-closing elements (e.g., </img>)

5. Error detections for malformed input

- Detects unmatched angle brackets (<, >) and invalid characters outside of tags.
- Provides user-friendly Value Error feedback when the structure is incorrect.

6. Read Command for Viewing File Content

- The read <filename> command allows users to view the content of a file before validation.

7. Case-Insensitive Tag Comparison

- It handles upper and lowercase differences in tag names to prevent false mismatches in the system.

8. Built-in Self-Closing Tag Rules

- This includes a list of standard self-closing HTML tags (e.g., img, br, hr) to prevent misuse. This approach tried to make the system more accurate.

9. Custom Error Handling with Clear User Prompts

- Friendly prompts like “Good!” or “Sad!” accompany success/failure messages to enhance usability.

10. No External Libraries Required (Pure Python)

- This system only uses standard Python libraries like re and os for maximum portability and simplicity in our system.

3.5 Working Mechanism

Step 1: User Input

The user can provide either a raw HTML/XML string or a filename with extension .html or .xml, that contains valid HTML/XML data content.

Step 2: Read Input

The system reads either plain user content from the user's input or whole file data having extension .html or .xml file directly given to the system.

Step 3: Tag Extraction

By using regular expressions, all valid tags are identified and extracted from the input.

Step 4: Tag Classification

Each tag is classified as:

- Opening tag (e.g.,)
- Closing tag (e.g.,)
- Self-closing tag (e.g., <img />)

Step 5: Stack-Based Processing (PDA Logic)

A stack is used to track nesting:

- Opening tags are pushed onto the stack.
- Closing tags are checked against the top of the stack and popped if matched.
- Self-closing tags are skipped.

Step 6: Validation Result

- If any mismatched or unexpected tags were found or detected by our system, then an error message is shown by the system.
- If all tags are matched and the stack is empty at the end, then this system will declare valid input.

3.6 Language and Tools

- Python 3
- Regular Expressions (re module)
- Command-line interface

3.7 Execution Instructions

Step 1: Run this command in terminal to start this project

$ python3 validator.py

```
(base) nabinpc@nabinpc:~/Desktop/Toc final project$ python3 validator.py
html/xml validator using pda
commands:
  validate <filename> - validate html or xml file content
  read <filename>     - display file content
  <html>...</html>    - you can also paste html directly
  exit                - quit the program


Enter HTML/XML:
```

Step 2: Enter plain html code

Enter HTML/XML : hi

```
(base) nabinpc@nabinpc:~/Desktop/Toc final project$ python3 validator.py
html/xml validator using pda
commands:
  validate <filename> - validate html or xml file content
  read <filename>     - display file content
  <html>...</html>    - you can also paste html directly
  exit                - quit the program


Enter HTML/XML: <div> hi </div>
Good! This is a valid html/xml structure.


Enter HTML/XML:
```

It will validate the raw HTML as expected.

Step 3: Validate HTML or XML file.

Syntax: validate <filename>

```
(base) nabinpc@nabinpc:~/Desktop/Toc final project$ python3 validator.py
html/xml validator using pda
commands:
  validate <filename> - validate html or xml file content
  read <filename>     - display file content
  <html>...</html>    - you can also paste html directly
  exit                - quit the program


Enter HTML/XML: validate note.xml
Good! This is a valid html/xml structure.


Enter HTML/XML: validate index.html
Good! This is a valid html/xml structure.


Enter HTML/XML:
```

Step 4: Exit the program

Directly type ‘exit’ to terminate the program

```
(base) nabinpc@nabinpc:~/Desktop/Toc final project$ python3 validator.py
html/xml validator using pda
commands:
  validate <filename> - validate html or xml file content
  read <filename>     - display file content
  <html>...</html>    - you can also paste html directly
  exit                - quit the program


Enter HTML/XML: exit
Thank you, for using PDA-Based HTML/XML Validator.
(base) nabinpc@nabinpc:~/Desktop/Toc final project$
```

3.9 Project Example

Example 1:

Input: Hello There!

Output: Good! This is a valid html/xml structure.

Example 2:

Input: validate index.html

File content : <html> <body> Hello World! </body> </html>

Output: Good! This is a valid html/xml structure.

CHAPTER 4 – TESTING

4.1 Results and Testing

To ensure the accuracy and robustness of this system, PDA-Based HTML/XML Validator, the program was tested many times with a diverse set of markup languages such as HTML and XML inputs. Test cases were designed to cover both valid and invalid structures, including well-formed document's structure, mismatched tags, missing opening and closing tags, improper nesting, and self-closing tags.

4.1.1 Test Case 1 (Raw Input)

Number of Test	Inputs	Observation / Output	Result
1	Test	Good! This is a valid html/xml structure.	Pass
2		Error at tag 3: closing tag does not match opening tag Error! This is a invalid or mismatched tags.	Fail
3		Error: unclosed tags keep remain on stack: ['div', 'div'] Error! This is a invalid or mismatched tags.	Fail
4	<img src="">	Error at tag 3: closing tag does not match opening tag <img> Error! This is a invalid or mismatched tags.	Fail
5		Good! This is a valid html/xml structure.	Pass

Table 1: Tase Cases

4.1.2 Test Case 2 (File Based)

Test 1: HTML file (index.html):

index.html

```
1  <html>
2    <body>
3      <img src="test.png" alt="Test Image" />
4      <p>Test Case 1</p>
5    </body>
6  </html>
```

Figure 2: HTML file with correct structure

Input and Result:

```
(base) nabinpc@nabinpc:~/Desktop/Toc final project$ python3 validator.py
html/xml validator using pda
commands:
  validate <filename> - validate html or xml file content
  read <filename>     - display file content
  <html>...</html>    - you can also paste html directly
  exit                - quit the program


Enter HTML/XML: read index.html

content of 'index.html':
<html>
  <body>
    <img src="test.png" alt="Test Image" />
    <p>Test Case 1</p>
  </body>
</html>



Enter HTML/XML: validate index.html
Good! This is a valid html/xml structure.


Enter HTML/XML:
```

Figure 3: Output for first test case

Observation: File is correctly parsed

Result: Pass

Test 2: XML file (note.xml):

note.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>MR</to>
  <from>Bin</from>
  <heading>Reminder</heading>
  <body>Don't forget TOC project assignment! </body>
  <date>Dute Date: 2025-05-04
</note>
```

Figure 4: XML file with incorrect structure

Input and Result:

```
(base) nabinpc@nabinpc:~/Desktop/Toc final project$ python3 validator.py
html/xml validator using pda
commands:
  validate <filename> - validate html or xml file content
  read <filename>     - display file content
  <html>...</html>    - you can also paste html directly
  exit                - quit the program


Enter HTML/XML: read note.xml

content of 'note.xml':
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>MR</to>
  <from>Bin</from>
  <heading>Reminder</heading>
  <body>Don't forget TOC project assignment! </body>
  <date>Dute Date: 2025-05-04
</note>




Enter HTML/XML: validate note.xml
error at tag 11: closing tag </note> does not match opening tag <date>
Error! invalid or mismatched tags.


Enter HTML/XML:
```

Figure 5: Output for second test case

Observation: File is correctly parsed but found a error

Result: Failed

4.2 Outcomes

Each and every test case was evaluated through both raw input and file-based validation. The system correctly identified structural errors and provided informative message which includes the location and nature of mismatches code structure. Additionally, the program was tested to ensure case insensitivity and graceful handling of malformed inputs. This also helps to prevent the system from crashing. The above testing confirms that the implementation reliably simulates PDA behavior and meets the expectations of a context-free language validator.

CHAPTER 5 – CONCLUSION AND FUTURE RECOMMENDATION

5.1 Conclusion

This project demonstrates how Pushdown Automata (PDAs) can be used to validate context-free languages like HTML/XML. It provides a clear example of automata theory applied to a real-world use case.

5.2 Future recommendations

- Future Work
- Stack visualization
- GUI/web-based version
- Attribute grammar or CFG validation

APPENDICES

Snippets of major source components

```
'''
Project Name: PDA-Based HTML/XML Validator
Theory of Computation (CSC720)
Class Project
'''

# import re module to use regular expressions
import re

# import os module to interact with the operating system for file handling
import os

# function to extracts all tags from the HTML or XML string
def tokenizeTags(html):

# extracting the valid html/xml tags using regex pattern matching
# matches opening tags (e.g., <div>), closing tags (e.g., </div>), and self-closing tags (e.g., <img/>)
tags = re.findall(r'</?[a-zA-Z][a-zA-Z0-9\-]*[^<>]*?/?>', html)

# filter out xml declarations (e.g., <?xml...>) and doctype declarations (e.g., <!doctype>)
filteredTags = [tag for tag in tags if not (tag.startswith('<?') or tag.startswith('<!'))]

# Check if no valid tags are found (e.g., input is plain text like "hello")
if not filteredTags:
raise ValueError("Error: no valid tags found in the input.")

# remove all valid tags from input to check for stray '<' or '>' characters
stripped = re.sub(r'<[^>]+>', '', html) # leaves only non-tag content

# checking if there are unmatched '<' or '>' in remaining content (e.g., "<div" without '>' or "hello>world")
if '<' in stripped or '>' in stripped:
raise ValueError("Error: unmatched '<' or '>' found outside of tags.")

# returning list of filtered tags for structural validation
```

```
return filteredTags

# this function identifies the type and name of each tag
def extractTagName(tag):

# check if it is a closing tag
if tag.startswith('</'):
return 'CLOSE', tag[2:].split()[0].rstrip('>')

# check if it is a self-closing tag
elif tag.endswith('/>') or tag[-2:] == '/>':
return 'SELFCLOSE', tag[1:].split()[0]

# otherwise, treat it as an opening tag
else:
return 'OPEN', tag[1:].split()[0].rstrip('>')

# defining a set of known self-closing tags in HTML
selfClosingTags= {
'img', 'br', 'input', 'meta', 'link', 'hr', 'col', 'area',
'base', 'embed', 'param', 'source', 'track', 'wbr'
}

# this function checks if tags follow proper nesting using PDA stack logic
def isWellFormed(tokens):

# initialize an empty stack
stack = []

# go through each tag
for i, tag in enumerate(tokens):

# extract tag type and tag name
tagType, tagName = extractTagName(tag)

# make tag name lowercase for case-insensitive conditions
tagName = tagName.lower()

# if it's an opening tag, push it to the stack
if tagType == 'OPEN':
stack.append(tagName)
```

```
# if it's a closing tag, check if it matches with the last opened tag
elif tagType == 'CLOSE':

# checking for self-closing tags which should not have closing tag
if tagName in selfClosingTags:
print(f"error at tag {i + 1}: </{tagName}> is a self-closing tag and cannot have a closing tag.")
return False

# if stack is empty to check if there's no opening tag to match
if not stack:
print(f"error at tag {i + 1}: unexpected closing tag </{tagName}> while stack is empty")
return False

# if the closing tag doesn't match the last opened tag then it can be say invalid
if stack[-1] != tagName:
print(f"error at tag {i + 1}: closing tag </{tagName}> does not match opening tag <{stack[-
1]}>")
return False

# if it matches then pop the tag from the stack
stack.pop()

# self-closing tags do not affect the stack

# after processing all tags
# checking stack because it should be empty for valid structure
if stack:
print(f"Error: unclosed tags keep remain on stack: {stack}")
return False


# if everything matched properly then return True
return True

# this function reads the content of a file
def readFromFile(filename):

# check if the file exists on the same path or directory
if not os.path.exists(filename):
print(f"Error: file '{filename}' not found, please check file and try agin.")
return None
```

```
# open the file and then read its content
with open(filename, 'r', encoding='utf-8') as file:
return file.read()

# this function for interactive PDA validator
def runValidator():

# show command instructions to the user for their ease
print("html/xml validator using pda")
print("commands:")
print(" validate <filename> - validate html or xml file content")
print(" read <filename> - display file content")
print(" <html>...</html> - you can also paste html directly")
print(" exit - quit the program")

# taking input
while True:
print('\n')

# read user input
userInput = input("Enter HTML/XML: ").strip()

# exit or terminate the program when user types 'exit' and then it will stop the loop
if userInput.lower() == 'exit':
print("Thank you for using!...")
break

# program for 'validate <filename>' command
if userInput.lower().startswith("validate "):
filename = userInput[9:].strip()
html = readFromFile(filename)

# continue if file not found
if html is None:
continue

# validate the html/xml structure
try:

# tokenize the html/xml content
tokens = tokenizeTags(html)
```

```
# print validation result
if isWellFormed(tokens):
print("Good! This is a valid html/xml structure.")
else:
print("Error! invalid or mismatched tags.")
except ValueError as e:
print(f"validation error: {e}")
print("Sad! invalid html/xml structure.")

# implementing 'read <filename>' to show content of the file
elif userInput.lower().startswith("read "):
filename = userInput[5:].strip()

# read the file content
content = readFromFile(filename)

# if content is not None then print the content
if content is not None:
print(f"\ncontent of '{filename}':")
print(content)
print()

# taking direct html/xml input from user
else:
html = userInput
try:

# tokenize the html/xml content
tokens = tokenizeTags(html)
if isWellFormed(tokens):
print("Good! This is a valid html/xml structure.")
else:
print("Error! This is a invalid or mismatched tags.")
except ValueError as e:
print(f"validation error: {e}")
print("invalid html/xml structure.")

# start the validator program
runValidator()
```