Compiler Design Lab Report

University of Chittagong

Compiler Design Lab, 2023

SP (Simple Parser)

Group 8

Authors:

Md. Masud Mazumder
Tonmoy Chandro Das
Tareq Rahman Likhon Khan
Md. Siam
Rabbi Hasan
Imtiaz Ahamad Imon
Md. Mustak Ahmed
Abdullah Al Faruque
Hasan Mia
Arif Hasan
Abu Noman Shawn Shikdar

Instructor:

Nasrin Sultana
Assistant Professor
Dept. of Computer Science & Engineering
University of Chittagong, Chittagong - 4331

Date: September 5, 2023

Contents

1 Overview

The SP (Simple Parser) is an indispensable tool in the realm of computational mathematics, meticulously crafted to tackle the intricacies of lengthy arithmetic expressions with precision and efficiency. This versatile parser extends its capabilities beyond the realm of simple calculations, empowering users to delve into the realm of complex mathematical computations.

One of its most notable features is its exceptional prowess in dealing with expressions laden with multiple operations and nested parentheses. As mathematical expressions grow in complexity, the need for a reliable and robust parsing tool becomes increasingly evident. SP rises to the challenge, effortlessly dissecting and evaluating expressions that involve addition, subtraction, multiplication, division, and intricate grouping through parentheses.

This report serves as a comprehensive guide, shedding light on the multifaceted nature of SP's capabilities. It navigates through the nuances of using SP to tackle extended arithmetic calculations, offering valuable insights into harnessing its power for mathematical analyses that extend beyond the ordinary.

In the following sections, we delve deeper into the requirements for utilizing SP, examine its strengths and limitations, and provide hands-on examples to illustrate its practical applications. Through this exploration, readers will gain a deeper appreciation for SP's ability to simplify complex mathematical challenges, making it an invaluable tool for mathematicians, scientists, engineers, and anyone who seeks to harness the power of computation to solve intricate problems.

2 Requirements

To effectively utilize the SP parser for extended arithmetic expressions, the following prerequisites are necessary:

- A Python 3.x installation on your system.
- A foundational understanding of arithmetic expressions.
- An arithmetic expression to evaluate, potentially with multiple operators and parentheses.

3 Pros & Cons

3.1 Pros

The pros of using a Simple Parser can vary depending on the specific type of parser and its intended use case. Below are some potential advantages of using a Simple Parser:

- **Versatility**: SP extends its capabilities to manage intricate arithmetic expressions with ease.
- **Operator Precedence**: It seamlessly handles operator precedence, ensuring accurate calculations.
- **Error Handling**: SP has implemented error handling to provide informative messages in case of issues.
- **Expandability**: Developers can further enhance SP's functionality for specific parsing requirements.
- **Ease of Implementation:** Simple parsers are often easier to implement compared to more complex parsing techniques. They use straightforward algorithms and may involve less code, making them accessible for developers who are new to parsing.
- **Clearer Error Reporting:** Simple parsers can provide clear and concise error messages when syntax errors are encountered in the input. This can be helpful for developers in identifying and fixing issues in their code.
- **Transparency:** Simple parsers are often easier to understand due to their straightforward code structure. This transparency can be beneficial when debugging or maintaining the parser.
- **Customization:** Developers can tailor simple parsers to their specific needs. This means they can focus on parsing only the parts of the language or document format that are relevant to their project.
- **Fewer Requirements:** Simple Parser is only dependent on Python which is available in most cases and thus it is easy to use.

3.2 Cons

- **Limited Parsing Power:** Simple parsers are not suitable for parsing complex grammars or languages with ambiguous syntax. They may struggle to handle languages with a high degree of complexity or context sensitivity.
- **Error Handling**: Error handling in simple parsers can be challenging. They may not provide detailed error messages or recover gracefully from syntax errors.

- **Maintenance Challenges**: As your parsing requirements become more complex, you may find that maintaining a simple parser becomes increasingly difficult. You might need to add more complexity to handle additional cases.
- **Limited Language Support**: Simple parsers are often limited in their support for various language features, which can be a significant drawback if you need to parse languages with advanced constructs.
- **Scalability Issues**: If your project grows, and you need to support more languages or handle larger inputs, a simple parser may not scale well and may require a complete rewrite.

4 Context Free Grammar (CFG)

Here we use this Context-Free Grammar (CFG) to demonstrate examples in the next sections. This is a very basic grammar of a simple calculator. This grammar can only handle four basic arithmetic operations +, -, *, /.

$$
\begin{aligned}
\langle expression \rangle &::= \langle term \rangle \mid \langle expression \rangle + \langle term \rangle \mid \langle expression \rangle - \langle term \rangle \\
\langle term \rangle &::= \langle factor \rangle \mid \langle term \rangle \cdot \langle factor \rangle \mid \langle term \rangle / \langle factor \rangle \\
\langle factor \rangle &::= \langle number \rangle \mid (\langle expression \rangle) \\
\langle number \rangle &::= \langle digit \rangle \mid \langle digit \rangle \langle number \rangle \\
\langle digit \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

5 Example (Working Examples)

Here's an example showcasing SP's prowess in evaluating long arithmetic expressions:

Algorithm 1: Arithmetic Expression Evaluation

Data: Arithmetic expression
Result: Result of the evaluation
while *true* **do**
  Read input expression;
  if *expression is "quit"* **then**
    Exit;
  else
    Parse the expression and evaluate it;
    Print the result;
  end
end

Listing 1: Code to evaluate arithmetic expressions

```
import re

# Function to evaluate arithmetic expressions
def evaluate_expression(expression):
    try:
        # Replace spaces for uniform expression formatting
        expression = expression.replace(" ", "")

        # Define a regular expression pattern to split the
           expression into parts
        pattern = r'(\d+|\+|-|\*|/|\(|\))'
        tokens = re.findall(pattern, expression)

        # Initialize stacks for operators and operands
        operators = []
        operands = []

        # Define operator precedence
        precedence = {'+': 1, '-': 1, '*': 2, '/': 2}

        # Function to perform calculations
        def apply_operator():
            operator = operators.pop()
            right_operand = operands.pop()
            left_operand = operands.pop()
            if operator == '+':
```

```
                result = left_operand + right_operand
            elif operator == '-':
                result = left_operand - right_operand
            elif operator == '*':
                result = left_operand * right_operand
            elif operator == '/':
                if right_operand == 0:
                    raise ZeroDivisionError("Cannot divide by
                        zero!")
                result = left_operand / right_operand
            operands.append(result)

        for token in tokens:
            if token.isdigit():
                operands.append(float(token))
            elif token in '+-*/':
                while (
                    operators and operators[-1] in '+-*/' and
                    precedence[operators[-1]] >= precedence[token
                        ]
                ):
                    apply_operator()
                operators.append(token)
            elif token == '(':
                operators.append(token)
            elif token == ')':
                while operators[-1] != '(':
                    apply_operator()
                operators.pop()  # Remove the '('

        while operators:
            apply_operator()

        # The final result should be on the top of the operand
            stack
        return operands[0]

    except Exception as e:
        return str(e)

# Main calculator loop
while True:
    expression = input("Enter an arithmetic expression (or 'quit'
        to exit): ")
    if expression.lower() == 'quit':
        break
    result = evaluate_expression(expression)
```

```
    print("Result:", result)
\end{python}
```

5.1 Examples and Results

- **Example: Addition**

  Input: 2 + 3

  Result: 5.0

- **Example: Subtraction**

  Input: 8 - 5

  Result: 3.0

- **Example: Multiplication**

  Input: 4 * 6

  Result: 24.0

- **Example: Division**

  Input: 10 / 2

  Result: 5.0

- **Example: Complex Expression with Parentheses**

  Input: ((3 + 5) * 2) / (7 - 3)

  Result: 4.0

- **Example: Division by Zero (Error Handling)**

  Input: 5 / 0

  Result: Cannot divide by zero!

- **Example: Multiple Operations with Operator Precedence**

  Input: 3 + 5 * 2 - 1

  Result: 12.0

- **Example: Quitting the Calculator**

  Input: quit

  (Exits the calculator loop)

The code will evaluate the expressions and display the results, handling simple arithmetic operations as well as error cases like division by zero.

6 Suggestion

- **Practice Required:** Proficiency in using SP for extensive arithmetic expressions demands practice and familiarity with operator precedence and parentheses usage.
- **Documentation Enhancement:** Enhance the documentation to include examples and detailed explanations for more complex use cases.
- **Further Expansion:** Consider future improvements, such as support for more advanced mathematical functions and mathematical constants.
- **Use of Alternatives:** There are a lot of alternative parser generator that are more advanced and can handle a lot complex CFGs. For example:
  - YACC
  - Bison etc.

  These advanced parser generator should be used to handle complex tasks.