Parser for a Simple Context-Free Grammar

What is a Parser?

A parser is a software tool that is used to verify the syntax of code or text based on a defined set of rules. In the context of computer science, parsers are commonly used to ensure that code written in a particular programming language adheres to the language's rules and grammar. In this report, we will be discussing the use of a parser to verify a context-free grammar (CFG) in our code. A CFG is a formal grammar that describes a set of strings in a language, and a parser can be used to check whether a given string adheres to the grammar's rules. By using a parser to verify the CFG in our code, we can ensure that our code will execute correctly and without errors.

Introduction:

This report discusses the development of a parser machine for a simple context-free grammar using the C++ programming language. The objective of this project is to create a tool that can parse a string of tokens and determine whether it conforms to the grammar rules of the language. This report discusses the techniques and tools used to develop this parser machine, as well as the timeline for completion.

Objectives:

The main objective of this project is to create a parser machine that can correctly identify whether a given string of tokens conforms to the rules of the context-free grammar. The machine should be able to identify any

syntax errors in the input and provide a meaningful error message to the user.

Tools and Techniques:

The parser machine was developed using the C++ programming language. The grammar rules for the context-free grammar were defined using the Backus-Naur Form (BNF) notation. The BNF notation provides a way to specify the grammar rules in a clear and concise manner.

The grammar rules were then written on an online calculator using BNF notation which generates sample inputs of that grammar. These sample strings were used to test the code whether the parser machine was programmed successfully or not. According to our results, these sample strings were successfully passed and when incorrect strings were written or if there were any syntax errors in the input, the parser machine returned an error message.

These are the sample inputs that were generated from online calculator which are Correct Structure:

(1) a
(2) (a)
(3) a+a
(4) a*a
(5) a*a+a
(6) ((a))
(7) (a)*a
(8) a+(a)
(9) a+a*a
(10) (a+a)
(11) (a*a)
(12) (a)+a
(13) a*(a)
(14) a+a+a
(15) a*a*a
(16) a*a*(a)
(17) a*a*a*a
(18) a*(a*a)
(19) a*a+a*a
(20) a*(a)*a

Parser Online Calculator:

Grammar

```
E → E + T
  | T.
T → T * F
  | F.
F → ( E )
  | a.
```

Some sentences generated by this grammar: {a, (a), a + a, a * a, a * a + a, ((a)), (a) * a, a + (a), a + a * a, (a + a), (a * a), (a) + a, a * (a), a + a + a, a * a * a, a * a * (a), a * a * a * a, a * (a * a), a * a + a * a, a * (a) * a}

- All nonterminals are reachable and realizable.
- There are no nullable nonterminals.
- The endable nonterminals are: F E T.
- No cycles.

nonterminal	first set	follow set	nullable	endable
E	(a	+)	no	yes
T	(a	* +)	no	yes
F	(a	* +)	no	yes

Pass Conditions:

- The given string must contain terminals/non-terminals that are present in the Grammar.
- The given string must have correct structure, i.e. no unclosed/open brackets, strings ending on operators, etc.
- String must be generated by the given Grammar.

Fail Conditions:

- Incorrect Input Structure.
- String contains terminals/non-terminals that are not present/generated by the Grammar.

Input Restrictions:

- No whitespaces/spacing in between characters. The code will fail to detect the string and terminate.
- Correct enclosing of all brackets (only round brackets “()” allowed) that are used.
- Correct Structure of using operators such as, “*”, “+”.

Conclusion:

The development of a parser machine for a simple context-free grammar using the C++ programming language was successful. The parser machine is able to correctly identify whether a given string of tokens conforms to the grammar rules of the language, and provides meaningful error messages in the event of syntax errors. The tools and techniques used in the development process proved to be effective in creating a reliable and efficient parser machine.

Correct Sample Output:

```
~~~ Grammar ~~~
E -> E + T | T
T -> T * F | F
F -> ( E ) | a

Input = (a*a)*a

E
T
T*F
F*F
E*F
(T)*F
(T*F)*F
(F*F)*F
(F*F)*a
(F*a)*a
(a*a)*a

~~~~ Correct Structure ~~~~
```

Incorrect Sample Output:

```
~~~ Grammar ~~~
E -> E + T | T
T -> T * F | F
F -> ( E ) | a

Input = (a+a)*

~~~~ Incorrect Structure ~~~~
```