Implementing Syntax analyzer For Compilation Process

Nwe Nwe Thant
Computer University, Monywa, Myanmar
nwelay01@gmail.com

Abstract

This paper is written to determine whether a string of tokens can be generated by a language grammar. The grammar specifies the structure of the language. Context free grammar is used to define the grammar of the input source language of a compiler. It is also known as Type-2 grammar. Syntax analysis can be done by two main techniques top-down and bottom-up. But almost all of the compilers used bottom-up technique. This papers is implemented as a syntax Analyzer for checking C++ program source code. In this paper uses bottom-up technique and Context free grammar to construct Syntax Analyzer. This system accepts C++ source code as an input and check whether there are syntax errors or not. If there are syntax errors, the system generate the error message with the suggested error line. If there are no syntax errors, the system generate the message there is no syntax errors.

keywords: Context free grammar, Bottom -up parsing theory, C++ program

1. Introduction

Software for early computers was primarily written in assembly language for many years but cost and complexity was dramatically increased for large software. Nowadays, software is primarily written in high level language by using an appropriate compiler.

Compiler is a program (or a set of programs) that translates text written in a computer language (the source language) into a another computer language (the target language). The original language code is also called as Source code and the translated language code or the output language code is also called as Object code [1].

The process of translating a source code language to a target language code is called compilation process. The compilation process can be divided into two parts namely analysis and synthesis.

In analysis stage, the source program breaks into constituent pieces and creates intermediates representations. It is a process to know what to do.

In synthesis stage, compiler generates the target program from the intermediate representations. It is a process translates what the programmer tells him into another equivalent code.

The analysis part can be divided into the following phases:

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis

The Synthesis part can be divided into the following phases:

1. Intermediate code generator
2. Code optimizer
3. Code generator

In Lexical Analysis, the program is considered as a unique sequence of characters. The Lexical Analyzer reads the program from left to right and sequences of characters are grouped into tokens. Token is the smallest unit of the program. The token syntax is typically regular language and so regular expression (regular grammar or type-3 grammar) can be used to recognize it

In Syntax Analysis, tokens are grouped into grammatical phrases represented by a parse tree which gives the hierarchical structure of the source program. To define the hierarchical structure of the source language, Context Free Grammar or type-2 grammar is used. This process is also called parsing. The component of a compiler doing parsing is called parser or Syntax Analyzer.

If Syntax Analyzer detects errors in syntactic structure of the source program, error message should be displayed [1].

2. Syntax analyzer

This paper**,** parsing, or, more formally, syntactic analysis, is the process of analyzing a sequence of tokens to determine their grammatical structure with respect to a given (more or less) formal grammar.

A parser is one of the components in compiler, which checks for correct syntax and builds a data

structure (often some kind of parse tree, abstract syntax tree or other hierarchical structure) implicit in the input tokens. This parses the source code of a computer programming language to create some form of internal representation. Programming languages tend to be specified in terms of a context-free grammar because fast and efficient parsers can be written for them. Parsers are written by hand or generated by parser generators. Context-free grammars are limited in the extent to which they can express all of the requirements of a language. Informally, the reason is that the memory of such a language is limited. The grammar cannot remember the presence of a construct over an arbitrarily long input; this is necessary for a language in which, for example, a name must be declared before it may be referenced. More powerful grammars that can express this constraint, however, cannot be parsed efficiently. Thus, it is a common strategy to create a relaxed parser for a context-free grammar which accepts a superset of the desired language constructs (that is, it accepts some invalid constructs); later, the unwanted constructs can be filtered out [2].

Once again the key to parsing involves finding a way to describe grammars. The method that has become totally dominant involves phrase structure grammars. The supposition is that the material to be parsed is a sequence of tokens (not just of characters) and that the parsing only depends on the types of these tokens not on any other attributes.

For instance, if a number can appear at some place then any number can appear. Thus an obvious start to defining a grammar is to list all the tokens that can possibly appear. These are referred to as Terminal Symbols. On then has another set of symbols, referred to as Non-Terminal Symbols. One non-terminal is identified as a Start Symbol. The grammar is completed by giving a set of re-write rules. Each of these (in the most general case) has a sequence of symbols (maybe some terminal and some non-terminal) on the left and another such sequence on the right.

There must be at least one non-terminal in the pattern in the left hand part of a rule. The set of sentences that make up the language defined by such a grammar can be obtained by starting with the start symbol. Then any time a sequence on the left of a rule appears in the string it can be replaced by the corresponding sequence on the right. If at some stage a string with no non-terminals arises then that string is part of the language.

If there is no way of generating a string of terminals, however many productions (the term usually used for what I just started to call re-write rules) are used then that string is not part of the language. It turns out to be interesting and sensible to consider various restrictions on the form of the productions. The first such set of conditions looks backwards in this course: insist that the left hand side of each production is just a single non-terminal, and each right hand side is either empty or consists of just a terminal followed by a non-terminal.

In general, grammars restricted in this severe way correspond very naturally to descriptions of non-deterministic finite automate (the non-terminals are states, and where there is a production with an empty right hand side that marks an accepting state). The next useful class of phrase structured grammars imposes just the constraint that the left hand side of each production should be a single non-terminal.

It can be determined that a language is not regular so no regular expression could define it and no finite automaton could accept it. Grammars where the left hand sides of productions are single non-terminals are known as context free grammars. A very similar style of result asserts that context free languages can be accepted by stack automata. These augment a finite-state control mechanism with a stack! Just as there is a pumping lemma for regular languages (and it can be used to show that certain languages are not regular) there is a pumping lemma for context free languages and again it can be used to show that some languages are not context free.

The usual example to quote is the language anbncn This consists of the strings abc, aabbcc, aaabbbccc and so on where in each case the number of occurrences of each letter match. Many texts on parsing discuss a class of grammars known as unconstrained phrase-structure grammars still generate languages. Once again they correspond to a model of computation: in this case Turing Machines. The fact that the ability to parse a general grammar of this form can call on the full power of general computation (and conversely any Turing machine program can have its behavior captured in a language defined by some phrase structure grammar) will have many undecidable problems around and that general grammars will not usually be sensible things to work with.

A final remark is that since there are computations that even a Turing Machine can not perform there will be languages that can not be defined at all phrase structure grammars. The most obvious example will be the language of “all programs in existing computer language X subject to the extra constraint that they must terminate”. And of course this would be a grammar, the compiler want to accept, so it could generate a diagnostic “this program may not terminate” for input that had that property! Many years of experience has shown

that context free grammars are powerful enough to express the structure that people want in programming languages, but constrained enough to be practical to work with. Some of these can be checked in later parts of a compiler. Before looking at detailed parsing techniques there is a slight area of conflict between the language theory approach and the practical one. The idea of mapping a grammar onto an automaton works in the context of accepting the languages.

This means it is directly valuable if a syntax checker reports a simple yes/no answer as to whether input text satisfies the rules of the grammar. However in most real compilers one wants more: a trace of which production rules were fired to get to the input sentence. These can be interpreted as a parse tree for the program. Extending a parser to build one of these is typically not that hard, but the big issue that this does raise is that of ambiguous grammars.

These are one where some particular sentence might be generated on two or more ways. In each case the parsing mechanism will accept the text, but there is no certainty which parse tree will be created. The simple approach to this is to restrict attention to grammars that are not ambiguous. A more complicated scheme allows grammars to start off ambiguous but provides ways of annotating the grammar to ensure that just one version of the parse can actually happen. A further messy area is that of error recovery. The beautiful theory of stack automata allows us to accept valid programs really rather efficiently. Again practical parsers will need to extend the theory with practical techniques (and sometimes tricks) to do better [3].

2.1. Types of parsers

There are two of parsers, Top-down parsing and bottom-up parsing. But this system is used bottom-up parsing theory.

Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules.

Bottom-up parsing -A parser can start with the input and attempt to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on. LR parsers are three categories, SLR (Simple LR),CLR(Canonical LR)and LALR (LookAhead LR) [6].But this system, LR parser are examples of bottom-up parsers and LR parsers will generate a rightmost derivation (although usually in reverse).Bottom-up Parsing is constructing that tree process from leaves to leaf. It produces right-most derivation and post-order tree traversal. It used left recursion in grammar.

2.2. Context free grammar

A context free grammar consists of terminals, non-terminals, a start symbol and productions.

1. Terminals are the basic symbols from which strings are formed. The word "token" is a synonym for "terminal".
2. Non-terminals are syntactic variables that denote sets of strings that help define language generated by the grammar. They impose a hierarchical structure on the language.
3. In a grammar one non-terminal is distinguished as the start symbol, and the sets of string it denotes is the language denoted by the grammar.
4. The productions of a grammar specify the manner in which terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal followed by an arrow (==>) followed by a string of non-terminals and terminals [5].

2.3. Parsing algorithm

Since the LR parser reads input from left to right but needs to produce a rightmost derivation, it uses reductions, instead of derivations to process input. That is, the algorithm works by creating a "leftmost reduction" of the input. The end result, when reversed, will be a rightmost derivation.

The LR parsing algorithm works as follows:

1. The stack is initialized with [0]. The current state will always be the state that is at the top of the stack.
2. Given the current state and the current terminal on the input stream an action is looked up in the action table. There are four cases:

a. a shift Sn:

The current terminal is removed from the input stream

the state *n* is pushed onto the stack and becomes the current state

b . a reduce Rm:

The number *m* is written to the outputstream

For every symbol in the right-hand side of rule *m* a state is removed from the stack

Given the state that is then on top of the stack and the left-hand side of rule *m* a new state is

looked up in the goto table and made the new current state by pushing it onto the stack

c. an accept: the string is accepted

d. no action: a syntax error is reported

3. Step 2 is repeated until either the string is accepted or a syntax error is reported [7].

2.4. Related work

Y.N. Srikant, Priti Shankar described a compiler translates a high-level-language program into a functionally equivalent low-level language program that can be understood and executed by the computer. Algorithms for compiler design teaches the fundamental algorithms that underlie modem compilers.[1]

Aho, Sethi, UIIman accepted result of the paper , Compiler technology is intimately intertwined with the target processor architecture, and compiler architects must solve new analysis and optimization problems to achieve the highest levels of performance.[2]

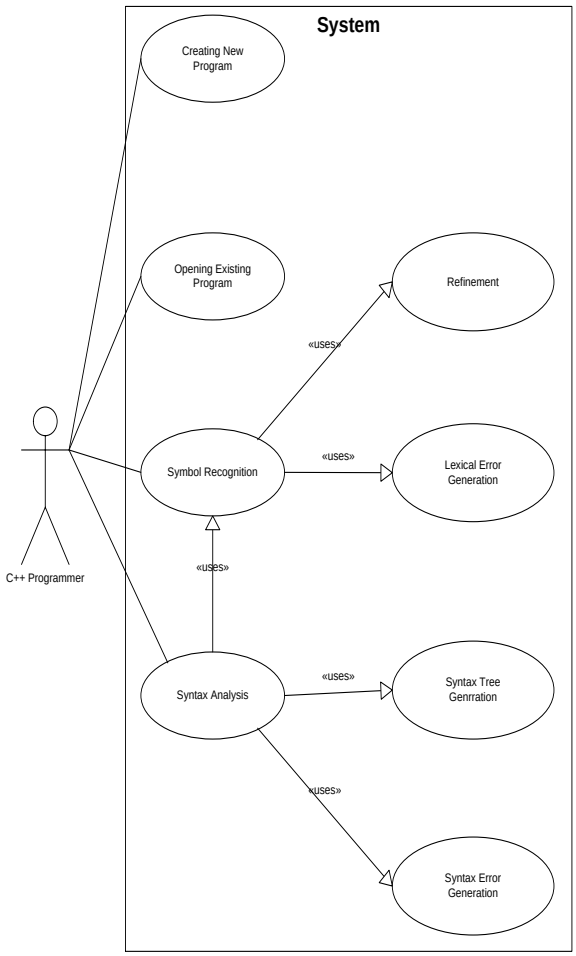
K. D.Cooper, Linda Torczon examined this paper describes how a particular notation was used to assist in the implementation of a Cobol compiler and of an interpreter for a simulation language.[6]

3. Use case diagram

C++ programmer can write a new C++ program or opening an existing C++ program. When the current C++ program is ready, programmer can check lexical analysis and syntax analysis by using Symbol Recognition process and Syntax Analysis process. When Symbol Recognition Process is called, the process accept input current C++ program and then pass it to refinement process to get rid of non necessary symbols (such as spaces, new line characters) in the current C++ program. Then Symbol Recognition Process generates the token each representing the words of the C++ program. If there is words which is not match any of token type is found in the C++ program, Lexical error generation process is called to show the Error Messages.

When Syntax Analysis process is called to check the current C++ program whether it has syntax errors or not, Syntax Analysis process is invoked and get the token streams from Symbol Recognition process. And then it checks the token stream whether it is validated with the context free grammar rules of the parser.

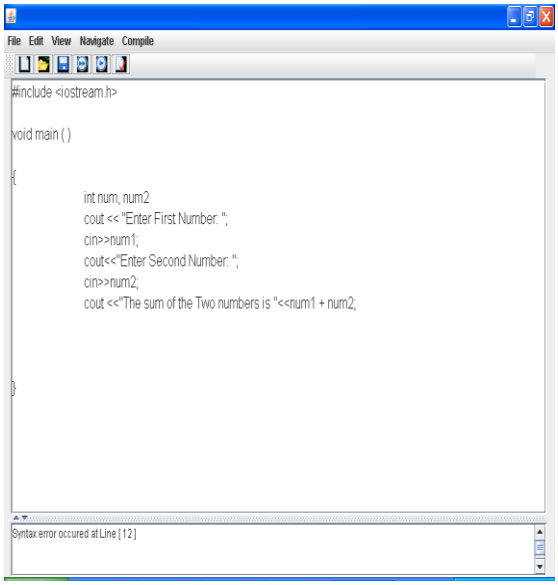
When the parser can produces the tokens stream which is equal to the output tokens stream of the symbol recognition process, the syntax analysis process determined the current C++ program has no syntax errors. If not the current C++ program has some syntax errors and called syntax errors generation to out put the error messages suggesting with the code line number.



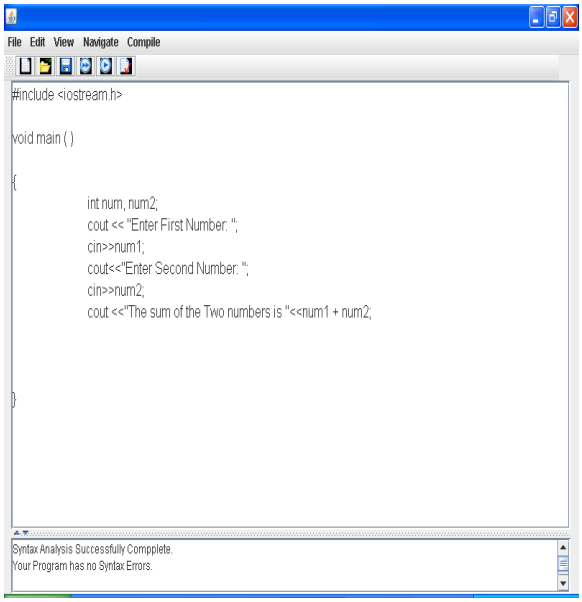
"Figure 3.2 Use Case Diagram for the System."

3.1. Showing for syntax analysis

Compile Menu has two menu commands namely Generate Tokens and Syntax Analysis. Generate Tokens menu command can be used to generate tokens consisting in C++. Source code program. Syntax Analysis menu command can be used to check whether C++ source code program has the appropriate structure. If there is syntax errors in program, it generate error message with suggestion line number. If there is not syntax errors in program, it generate the program has no syntax errors.



"Figure 4.9. Compile menu"



"Figure 4.10. Compile menu"

4. Conclusion

This paper is implemented as a syntax analyzer for C++ program source code. This program uses up to 100 rules and cup tool. This paper covers the front end portion of the compiler except Semantic analysis. It checks or validates the structure of C++ program. If there are syntax errors in the C++ program, the system generate error message with the suggested error line number. If there are no syntax errors, the system generate the message "There are no syntax errors in the source program".

5. Result

When the system checks a C++ program for the syntax errors checking, the system generate the message "There is no syntax errors in this program" if the program has no syntax errors. If the C++ program has syntax errors, the system generate the message "Syntax errors occurred at line number : linenumber " message with the suggestion of the errors at suggested line number.

6. Further extension

Any one who interested my paper and willing to extend this paper should extend with the following additional features:

1. The given C++ program can be translated to the associated parse tree.
2. The system can display Semantic Analysis along with Syntax analysis.
3. Attribute grammar can be used to do Syntax and Semantic analysis rather than Context Free Grammar.
4. Static Semantic Analysis such as Type Checking should be represented.
5. The current program writes java programming language, but uses other languages.

7. References

[1] Y. N. Srikant, Priti Shankar, "The Compiler Design Hanbook: Optimization and Machine Code Generation", 2003

[2] Aho, Sethi, Ullman ,*"Compiler: principle, techniques, and tools"* **,** 1986, ISBN: 9780201100884

[3] S. S. Muchnick, *"Advanced Compiler Design and Implementation"*, 1997

[4] S. Chattopadhyay, *"Compiler Design"*, 2006

[5] A. W. Appel, Jens Palsberg, *"Modern Compiler Implementation in Java"* ,1997, ISBN: 0 521 82060

[6] K. D.Cooper, Linda Torczon, *"Engineering a Compiler"*,2003

[7] O. G Kakde, *"Algorithms for Compiler Design"*, 2002