Design and Implementation of Software Evolution Process Description Language Lexical Analyzer

1st Jiang Na
School of Physics and Information EngineeringZhaotong University
Zhaotong, China
27805044@qq.com

2nd Gu Qingchuan
School of Physics and Information Engineering
Zhaotong University
Zhaotong, China
elaincoco@foxmail.com

3rd Yang Haiyan
School of Physics and Information Engineering
Zhaotong University
Zhaotong, China
gqch123456@163.com

***Abstract*—Software evolution is a very important form in the software life cycle. It is the process of software system constantly adjusting itself to meet the changes of the Internet open environment and user needs. It uses the Software Evolution Process Description Language (EPDL) to describe and extend the meta-model ability of the software in its evolution process, and describe the computer morphology of the software evolution process. According to the lexical rules of EPDL language, the design model of EPDL lexical analyzer is proposed in this paper –Extended Backus–Naur Form (EBNF) paradigm describing word rules, finite state automata recognizing words, and finally implementing lexical analyzer to complete the task of lexical analysis of source code.**

Keywords—software evolution, EPDL, compilation process, lexical analysis

I. INTRODUCTION

In order to adapt to the changing environment and user needs, software systems need to be constantly adjusted. Software evolution is a very important form in the software life cycle. Software evolution process is the workflow in software evolution. Its purpose is to establish the overall task framework of software evolution and improve the efficiency of software evolution[1]. In the book Modeling of Software Evolution Process, through the comparison of the different characteristics of heritage software system evolution and traditional software engineering, the method of establishing software evolution process model is proposed, and the software evolution process model supporting process reuse and continuous evolution of legacy software is constructed. According to the needs of software evolution, the software evolution process description language is defined on the basis of the software evolution process meta-model EPMM. It is a computer modeling language used to describe the software evolution process. Usually, after a software evolution process model being established, EPDL is used to describe and expand its details[1]. In this paper, the software evolution process description language EPDL (Evolution Process Description Language) proposed in the book Modeling Software Evolution Process briefly introduces the lexical analysis process of the language at compile time. As the foundation of design and implementation of the software evolution process description language compiler, it has good platform and scalability.

II. BRIEF INTRODUCTION OF EPDL

In the book Modeling of Software Evolution Process, in order to analyze the individualization requirements of software evolution, a formal software process model representation tool EPMM is designed to be used for the formal representation of the evolution process of heritage software. An object-oriented software evolution process description language, EPDL, is designed to describe the computer-formed modeling language of software evolution process. Some of the EBNF paradigm definitions of EPDL language are selected as follows (software process definition):

<Software Process>::=PROCESS<Software Process Name>[**FROM**<Software Process Name>][**TYPE**<Type Definition List>;][**PACKAGE IMPORTS**<Variable Declaration List>**EXPORTS**<Variable Declaration List>;**LOCALS**<Variable Declaration List>;**ENTRANCE**<Activity name>;**EXIT**<Activity name>;**MINI SPECIFICATION**<Mini Specification>;**KEY WORDS**<KeyWords>][<Activity List>]

BEGIN

[**CONDITION SET**<Condition Assignment statement List>;][**ACTIVITY SET**<Activity Assignment Statement List>;][**ARC SET**<Arc Assignment statement List>;][Initial Marking]

END; [1]

III. COMPILATION PROCESS

For the source code written in the programming language EPDL, to apply to the actual, the computer must recognize and execute it. And the two ways below can be adopted: compile it into a low-level language (machine language or assembly language) and then execute it; or convert it to other high-level languages, which are then compiled into machine language for execution by the corresponding high-level language compiler. No matter which compilation or conversion method is used, it is inseparable from compilation.

Compilation is the process of converting a high-level language program into a low-level language program and then executing it on a computer. In the process of compiling, according to the purpose and implementation ways of each stage, it can be divided into six stages: lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and object code generation. Together with table management and error processing, the whole compilation process is formed.

  DOI: 10.25236/iccpb.2018.019

This paper only analyzes and designs the lexical analysis stage in the compilation process.

IV. LEXICAL ANALYSIS

As the first stage of compilation process, the main task of lexical analysis is to process the source program at the level of word. The word is the smallest grammatical unit with real meaning in high-level language. It consists of characters and is meaningful and cannot be subdivided[3]. After the source program is split into words, they are classified and the number of lines is recorded. When the source code has an error, the compiler indicates the error location to the user.

Lexical analysis uses the finite state automaton as the theoretical basis, and uses EBNF paradigm to describe the word lexical rules. Generally, if a word conforms to the lexical rule of a language, the word is said to be the correct word that conforms to the lexical rule, otherwise the word is considered to be the wrong word. The finite state automaton is a device used to identify whether a word conforms to a lexical rule, and it provides a method and tool for constructing a lexical analysis program. The finite state automaton has a finite number of internal states, with one input and one output. There are several states between input and output. The system determines the later operation of the system based on the current state and the input characters it faces. The finite state automaton model is shown in Fig. 1:

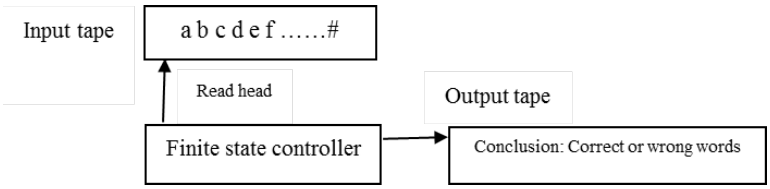

Fig.1 Finite state automaton model

The finite state automaton has an input tape for storing the word to be recognized, and an output tape for displaying whether the word satisfies the lexical rule. The read head of the finite state automaton is used to read the characters constituting the word on the input tape, and its state is composed of an initial state, an intermediate state, and a termination state. Before the word is recognized, the finite state automaton is in the initial state, and the read head is zeroed; when the word is recognized, the read head moves to the right, and after reading a certain character, the state is converted from the initial state to an intermediate state until the last character of a word is read. If the word is read and the state reaches the final state, it indicates that the word to be recognized conforms to the lexical rule.

Based on the finite state automata theory, the EBNF paradigm grammar definition of EPDL language in the Modeling of Software Evolution Process. It can be said that the words constituting the EPDL language can be divided into keywords, identifiers, operators, delimiters, constants. Use the EBNF paradigm to describe lexical rules:

<keywords>::=FOR | { | } |VAR | PROCEDURE | BOOLEAN | UNION | SEQ | INTEGER | STRING | REAL | IF | THEN | ELSE

<identifier>::=<letter>{< letter>|<number>} // user-defined process name, activity name, task name, etc.

<operator>::= + | - | . | * | / | < | > | = |!= | == | ++ | --

<delimiter>::= , | { | } | (|) | # | : | ;

<constant>::=< unsigned number>|< unsigned number>.< unsigned number>

< unsigned number >::=<number>{<number>} // integer/decimal in the source code

<letter>::= a | b | ……| X | Y | Z

<number>::= 0 | 1| 2 | ……| 9

Based on the above analysis, the EBNF paradigm is used to describe the lexical description of each type of word, and the state transition diagram of the lexical rule is constructed as follows [4]:

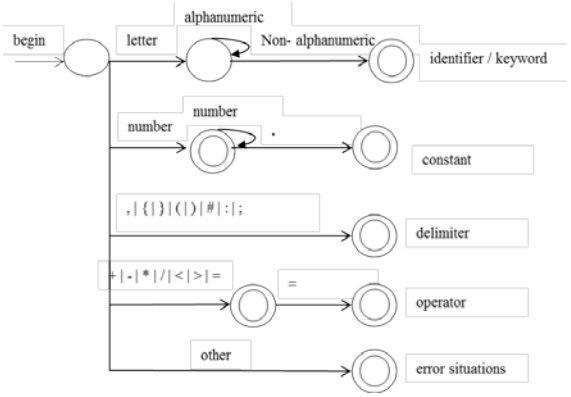

Fig.2 EPDL lexical rule state transition diagram

Based on the finite state automaton theory, a finite state automaton can be generated. After the deterministic and minimization algorithm is used, the ε arc is eliminated and processed by the computer to analyze the words in the EPDL source program. Analyze the program written in a specific EPDL code, lexical analysis results (type, word, number of lines) are shown in Figure 3:

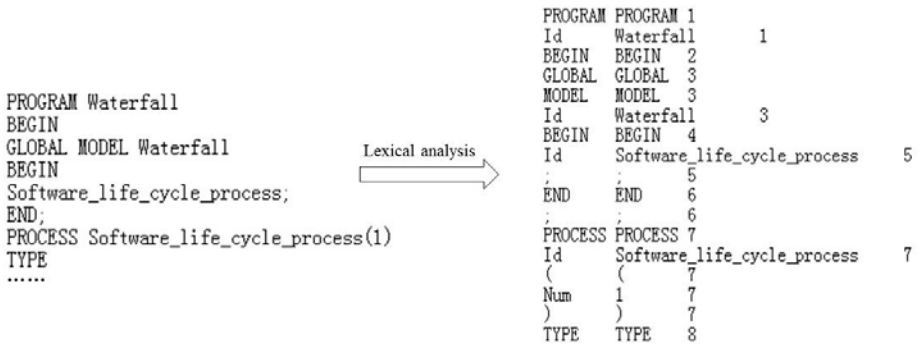

Fig.3 Process of lexical analysis

V. CONCLUSION

The EPDL language is a computer language that takes formal modelling for the evolution process and must be compiled to make it practical. This paper describes the analysis design method in the EPDL lexical analysis. Based on the EBNF paradigm grammar definition of EPDL language, combined with the finite state automata theory, the EBNF paradigm is used for paradigm description of the word lexical rules. The EPDL finite state automaton is constructed to determine that the minimum finite state automaton is used to implement the lexical analysis function. The design and implementation of lexical analysis provides input data for grammar analysis, and at the same time contributes to the implementation of the later EPDL compiler, providing a theoretical basis for future design work.

REFERENCES

[1] Tong Li. An Approach to Modelling Software Evolution Processes[M]. Springer and Tsinghua University Press. 2009.

[2] Jiang Na, Kong Hao. Design and Analysis of a Compiler Supporting Software Evolution Process Description Language[J]. Journal of Mianyang Normal University, 2013, 32(2): 99-102.

[3] Wang Shengyuan, Dong Yuan, Zhang Suqin, Lu Yingzhi, Jiang Weidu. Compiler Principles (Third Edition) [M]. Tsinghua University Press. 2015.

[4] Jiang Na. Design and Implementation of JAVA-based EDPL Compiler [D]. Yunnan: Yunnan University, 2010.