Method Selection in EPDL Parsing

Jiang Na [1,a], Yang Haiyan [1,b], and Gu Qingchuan [1,c]

[1]School of Physics and Information Engineering Zhaotong University Zhaotong, China

[a]27805044@qq.com; [b]elaincoco@foxmai.com; [c]gqch123456@163.com

Keywords: EPDL; Compilation process; Parsing; Parsing method

Abstract: Parsing is a very important stage in the compilation process. On the basis of lexical analysis, combining with the defined grammar rules, it's a process to analyze whether the words can form a grammatical unit that conforms to the grammatical rules. By analyzing the advantages and disadvantages of various parsing methods, the recursive descent analysis program is selected as the parser of the Software Evolution Process Description Language (EPDL). According to the grammar rules of EPDL language, this paper proposes a design model of EPDL parser - context-free grammar describes grammar rules, and push down automata recognition statements.

1. EPDL Introduction

The Evolution Process Description Language (EPDL) is a formal modeling language used to describe the evolution process of software in the book An Approach to Modelling Software Evolution Processes. It is an object-oriented high-level programming language. Using the EPDL language modeling software evolution process, the software evolution process can be dynamically presented, supporting concurrency, iteration, and integration. [1]

As an emerging language, the EPDL source code needs to be converted into an equivalent C language data structure and finally gets a visual Petri net. This conversion process is inseparable from compilation. In this paper, the common parsing methods in compiling are compared. The EPDL parser is analyzed and designed, which lays a foundation for design and implementation of post-semantic analysis and intermediate code generation. It has good platform and extensibility.

2. Compilation Process

Compilation is the process of translating a high-level language program into a low-level language. As a new formal language, EPDL goes through the following stages in the compilation process [2]:

2.1. Lexical Analysis.

Using the source program as input, scanning and decomposing the source program, and breaking it into words.

2.2. Parsing.

Based on the results of lexical analysis and defined grammar rules, analyze whether the identified words constitute a recognized sentence that satisfies the grammatical rules.

2.3. Semantic Analysis.

Based on parsing, identify the meaning of the sentence.

2.4. Intermediate Code Generation.

According to the meaning of the sentence, make the initial translation for the code and get the intermediate code between the source code and the target code.

2.5. Code Optimization.

Optimize the intermediate code for the purpose of improving efficiency.

  DOI: 10.25236/isrme.2019.012

2.6. Object code Generation.

Convert optimized intermediate code into low-level language code on a specific machine.

2.7. Form Management.

Record various information of the source program in the above six stages and various conditions during the compilation process.

2.8. Error Handling.

If there is an error in the code, try to prompt the user for the location of error and the type of error. This paper only compares, selects, and designs the methods involved in parsing.

3. Overview of Parsing

Parsing is one of the core functions of the compiler [3]. Its task is to determine whether the sequence of words sent by the lexical analyzer can form grammatical units (phrases, clauses, statements, process, procedures) that conform to grammatical rules. Generally, the grammar rules of a language are described using a context-free grammar, and with the help of the push-down automaton, the sentence to be recognized is judged whether it conforms to the grammatical rules of the language. When the high-level language program written by EPDL has a syntax error in parsing, the parser indicates to the user where the error occurred and the cause of the error.

In the book *An Approach to Modelling Software Evolution Processes,* the grammar rules of EPDL language are completely defined, and some definitions are selected as follows (EPDL program definition):

<EPDL Program>::=PROGRAM <Program Name> [<Glossary>;] BEGIN <Global Model> <Software Process List> END.

<Glossary>::=<Term>|<Term>; <Glossary>

<Term>::=#define <Term Name>: <Term Explanation>

<Software Process List>::=<Software Process>|<Software Process> <Software Process List>[3]

In parsing, if the sentence to be recognized conforms to the grammatical rules of a language, the sentence is said to have the correct syntax, otherwise the syntax is wrong. The pushdown automaton is used to identify whether a sentence conforms to the grammatical structure of a grammar, and it provides methods and tools for constructing a parser. The pushdown automaton has an input tape for storing the sentence to be recognized and an output tape for displaying whether the gramma is a correct conclusion. In the parsing, the read head sequentially reads the words of the sentence to be recognized from the left to the right, and the pushdown stack is used to determine the next step of operation of the parsing. Its operation is determined by the word pointed to by the read head and the top symbol of the stack. The pushdown automaton model is shown in Figure 1:

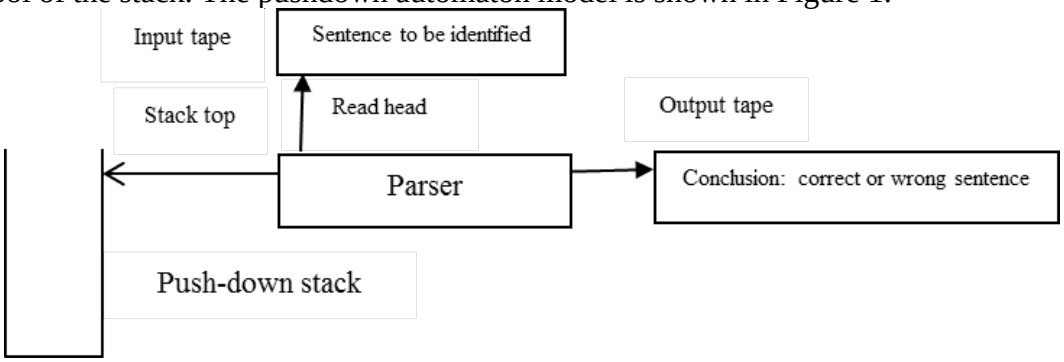

Fig. 1 Push-down automaton model

4. Comparison and Selection of Parsing Methods

The book uses the EBNF paradigm to describe EPDL grammar rules, which define the definitions that conform to context-free grammars, that is, there is one and only one non-terminal character in

the left part of each production. There are two ways analyzing whether the sequence of words obtained by lexical analysis of EPDL meets the grammatical rules:

4.1. Top-down parsing method (derivation)

It is the process starting with the start symbol <EPDL Program>, using the right part of the production to replace the left part, and finally generating the sentence. The main problem is to choose which production method to use for derivation. The grammar is required to be an LL (1) type grammar, reading the input string from left to right and constructing the leftmost derivation, and reading the next symbol of the read head for judgment.

1) *Recursive descent LL (1) analysis method:* Easy to implement, more intuitive, but recursive calls may result in lower efficiency.

2) *Table-driven LL (1) analysis method:* It constructs a predictive analysis table for the grammar, and determines the next operation according to the current top-of-stack symbol and the read-head symbol by the predictive analysis program.

4.2. Bottom-up grammar analysis method (stipulation)

It's the inverse process of derivation, starting from the sentence in the EPDL source program, using the left part of the production to continuously replace the right part, and finally reaching the starting symbol <EPDL Program>. The main problem is to choose which symbol string to use for the specification.

1) *Simple priority analysis method:* The grammar is required to be a simple priority grammar, a normative stipulation method, and a stipulation in strict accordance with the handle. The simple priority analysis table constructed is based on the priority relationship of all symbols, so the analysis table is large, although it is standardized, the efficiency is low, and the actual use value is not large.

2) *Operator-first analysis method:* The grammar is required to be the operator-first grammar, which is not the normative stipulation method, and is stipulated according to the left-most prime phrase. The constructed operator-first analysis table only considers the priority relationship between the terminators, so the analysis table is smaller than the simple priority table of the same grammar, the efficiency is higher, and the analysis speed is faster, as long as it is used for the analysis of expressions.

3) *LR analysis method:* It consists of a general control program, an analysis table, and an analysis stack. The input string is read from left to right and the inverse process of the rightmost derivation is constructed, that is, the leftmost stipulation. According to different ways of finding handles, it can be divided into LR (0), SLR (1), LR (1), LALR (1). The LR analysis method can basically identify all context-free grammars and has strong analysis capabilities, thus it can accurately locate the error location, but sometimes the implementation cost is too high.

By comparing the five grammar analysis methods, the advantages and disadvantages of the various methods are shown in Table 1:

Table 1. Advantages and Disadvantages of the Parsing Method

Analysis method	Normative or not	Size of analysis table	Actual value	Grammar requirements / scope of application
Recursive descent LL(1) analysis	Yes	None	High	LL(1) grammar/applicable range is slightly larger
Table driven LL(1) analysis	Yes	Small	High	LL(1) grammar/applicable range is slightly larger
Simple priority analysis	Yes	A little big	Low	Simple priority grammar / minimum scope of application
Operator precedence analysis	No	Small	High	Operator priority grammar / small scope of application
LR analysis	No	Big	High	Context-free grammar / large scope of application

From the definition of EPDL grammar rules in the book, the grammar is a context-free. It's easier to design, control, and implement by using the recursive descending LL (1) analysis method. Partial production contains left common factors, such as:

<Message Name List> → <Message Name>[(<Variable List>)]|<Message Name>[(<Variable List>)], <Message Name List>

<Variable List>→ <Variable Name>|<Variable Name>, <Variable List>

<Receiver List>→<Receiver>|<Receiver>,<Receiver List>

<Parameter Set>→<Expression>|<Expression>, <Parameter Set>[1]

The left common factor is extracted to perform equivalent transformation on the production. The specific steps are shown in Figure 2:

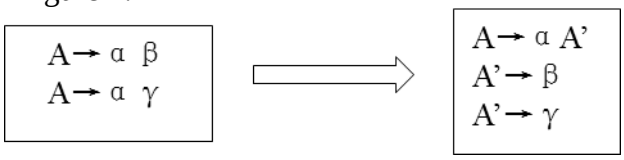

Fig. 2 Extracting the left common factor [3]

Some grammars contain left recursion, such as:

<Code Segment>::=<Code Segment>;<Code Segment>

Eliminate the left recursion and perform equivalent transformation on the production. The specific steps are as shown in Figure 3:

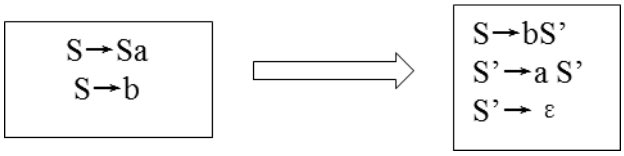

Fig. 3 Elimination of left recursion [3]

The processed grammar proves to be an LL (1) grammar, and a recursive descent analysis can be used to construct a grammar parser. Taking the EPDL program definition excerpted in the third part as an example, the recursive analysis subroutine corresponding to <EPDL Program> can be described as:

```
void ParseEpdlProgram()
{
MatchToken(PROGRAM);
ParseProgramName();
for (i=0; First(<Glossary>)[i]!='\0';i++)
        {
        if (lookahead==First(<Glossary>)[i])
        {
             ParseGlossary();
             MatchToken(;);
        }
  }
  MatchToken(BEGIN);
ParseGlobalModel();
Parse SoftwareProcessList();
MatchToken(END);
MatchToken(.);
}
```

An analysis subroutine is constructed for each non-terminal character of the grammar, and the parser begins with an analysis subroutine that invokes the grammar start symbol. Taking a simple EPDL program as an example, the results of analysis using the recursive descent parser are shown in Figure4:

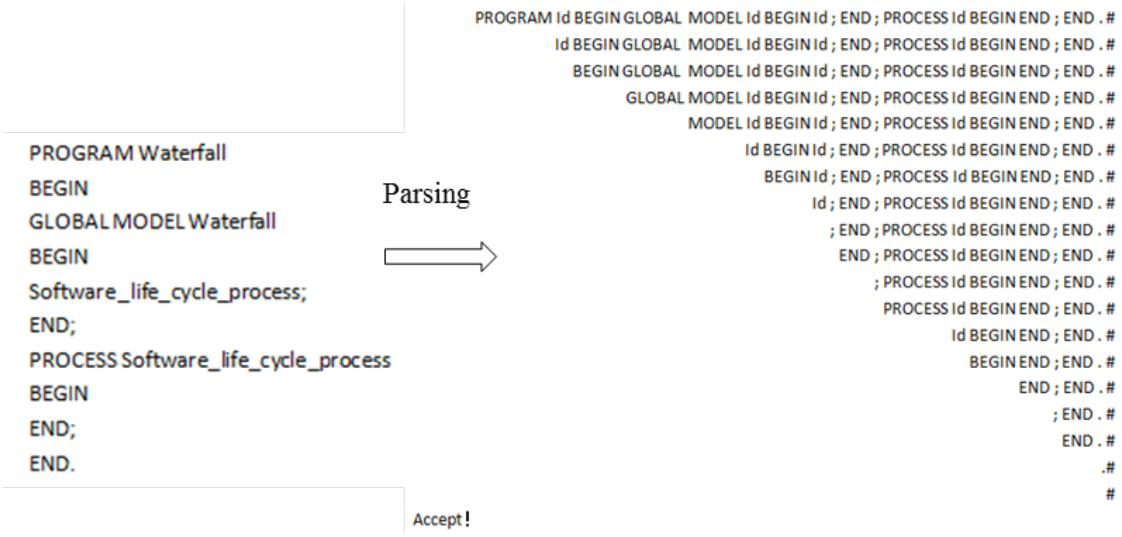

Fig. 4 Parsing process

5. Conclusion

This paper introduces the parsing task of software evolution process description language, and analyzes the characteristics, advantages and disadvantages of different grammar analysis methods. Combined with the grammatical definition of the EPDL language, the recursive descent analysis method is chosen to construct the parser, and the specific grammar processing method and implementation method are given. Through the analysis and design of EPDL parser, the recursive descent analysis method is used to design a grammatical Tokenizer that meets the requirements, which provides a basis for the late semantic analysis, intermediate code generation and the overall implementation of the compiler.

References

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

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

[3] S.Y. Wang, Y. Dong, S.Q. Zhang, Y.Z. Lu and W.D. Jiang. Compiler Principles (Third Edition) [M]. Tsinghua University Press. 2015.

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