ANTLR을 이용한 MEGACO 파서의 개발

황의윤[0]* 허정석* 김성규** 이명준*
울산대학교 컴퓨터정보통신공학부*
(주)온넷기술**
{heyoon[0], heojs, mjlee}@ulsan.ac.kr*
sungkim@ont.co.kr**

Development of a MEGACO Parser using ANTLR

Euiyoon Hwang[0]*, Jeongseok Heo*, Sunggyu Kim**, Myungjoon Lee*
School of Computer Engineering & Information Technology, University of Ulsan*
Onnet Technologies**

요 약

MEGACO(MEdia GAteway COntrol) 프로토콜은 VoIP(Voice over IP) 시스템에서 MGC(Media Gateway Controller)와 MG(Media Gateway)간에 통신을 정의하는 표준이다. MEGACO 명세서에는 통신규약에 대한 내용을 ABNF(Argumented BNF)형식으로 제공하고 있으나, 이것을 그대로 사용하여 MEGACO 메시지를 분석하는 파서(Parser)를 개발하기에는 많은 어려움이 있다. 규칙(rule)과 규칙간의 비결정적인 요소와 토큰과 토큰간의 모호성이 많이 존재하기 때문에 적절한 변환을 통하여 파서를 제작하여야 한다.

본 논문에서는 ANTLR 파서 생성기와 MEGACO 명세서에서 제공되는 ABNF문법을 사용하여 MEGACO 파서(Parser)를 개발하였다. ANTLR에서 제공하는 Syntactic predicate와 Semantic Predicate등을 적절하게 사용하여 명세서에 존재하는 여러 가지 형태의 비결정적인 구문과 모호한 토큰들을 제거 하였다.

1. 서 론

1990년대 중반이후 전 세계적으로 인터넷의 광범위한 확산과 더불어 음성데이터 분야에도 인터넷을 통해서 서비스를 제공하기 위한 기술들이 나타나기 시작했다. 이러한 개념으로 인터넷상에서 음성데이터 서비스를 제공하기위한 *VoIP(Voice over IP)* 기술[1]이 등장하게 되었다. 그러나 초기의 VoIP기술은 인터넷상에서 제한적이고 통화 품질 또한 만족할만한 수준이 아니었다. 이에 따라 기존의 PSTN과 새로운 패킷교환 망 사이에서 서로 다른 Call signaling 프로토콜의 연동과 다양한 종류의 미디어 스트림(media stream) 형식 변환이나 전송 등의 새로운 기능을 담당해야 할 게이트웨이(gateway)가 필요하게 되었다. 그리고 이런 기능은 PSTN망에서와 마찬가지로 제어망과 실제 미디어 스트림 전송을 담당할 전송망을 분리하는 방향으로 표준화가 진행되었다.

MEGACO(MEdia GAteway COntrol)[2]프로토콜은 이러한 제어망(MGC-Media gateway control)과 전송망(MG-Media gateway)간의 통신 방법을 정의하는 표준이다. MEGACO 명세서에는 서로간의 통신 규약을 ABNF(Argumented BNF)형식으로 정의하고 있지만, MEGACO에 대한 파서(Parser)를 제작하기 위해 사용하기에는 많은 어려움이 있다. 이는 규칙(rule)과 규칙간의 비결정적인 규칙과 토큰(Token)과 토큰간의 모호성이 많이 존재하기 때문에 적절한 변환을 통하여 파서를 제작하여야 한다. 파서를 제작하는 도구는 Lex[3]와 Yacc[4], Flex[5]와 Bison[6], 그리고 ANTLR[7] 등이 있다.

본 논문에서는 ANTLR 파서 생성기(Parser Generator)와 MEGACO 명세서에서 제공하는 ABNF의 통신규약을 사용하여 MEGACO 파서를 개발하였다. 위에서 말한 여러 가지 비결정적인 요소들과 모호한 부분들을 제거하기 위해 ANTLR에서 제공되는 고급 기술들을 사용하였다.

본 논문의 2장에서 ANTLR에 대해서 간략히 살펴보고, 3장에서는 ANTLR에서 MEGACO 파서의 정의에 대해서 살펴본다. 4장에서는 파싱한 결과를 위한 파서 트리에 대해 알아보고, 마지막으로 5장에서는 결론 및 추후연구에 대해 기술한다.

2. ANTLR

ANTLR(ANother Tool for Language Recognition)은 Java, C++, C#등으로 이루어진 문법 표현으로부터 파서를 생성하는 도구이며, Predicted LL(k) 파싱을 한다. ANTLR은 이해하기 쉽고, 효과적이며, 유연하기 때문에 많은 사람들이 사용하고 있다.

ANTLR을 사용하기 위해서는 문법 파일을 정의하여야 하는데, 이 파일에는 입력된 문자열을 토큰으로 분리하는 스캐너(Scanner)와 토큰으로부터 정의된 규칙에 올바른 것인지를 판단하는 파서에 대한 표현을 한꺼번에 정의할 수 있다.

2.1. ANTLR 문법 형식

ANTLR 문법 파일은 Header, Tokens, Options, Lexer class, Parser class등의 부분으로 구성되며, 아래의 [표1]에 나타난 형태로 정의된다.

```
header { … }
options { options for entire grammar file }

{ optional class preamble }
class LexerClass extends Lexer
options { … }
tokens...
lexer rules...

{ optional class preamble }
class ParserClass extends Parser;
options { … }
tokens...
parser rules...
```

[표1] ANTLR 문법 파일의 구성

* 본 논문은 (주)온넷기술의 지원으로 수행되었음

3. MEGACO 파서의 정의

ANTLR에서 MEGACO 파서를 생성하기 위해서는 2장에서 소개한 문법 파일을 정의해야 한다. 우선, MEGACO 명세서에서 제공하는 ABNF문법을 ANTLR에서 지원하는 EBNF문법으로 변환해야 한다. 그리고 변환된 EBNF문법에서 렉서(Lexer)와 파서에 대한 것을 분리해서 정의해야 한다.

3.1. ABNF의 변환

ABNF와 EBNF는 서로 동일한 의미를 가지는 부분이 많이 존재하기 때문에 대부분의 규칙들은 ABNF의 의미에 맞는 EBNF 표현을 사용하여 변환이 가능하다. 그러나 변환된 EBNF를 ANTLR에서 사용하기 위해서는 많은 비결정적인 규칙과 모호한 토큰을 수정하여야 하며, 자세한 사항은 아래에서 차례로 언급한다.

3.2. Lexer class의 정의

렉서는 입력 파일로부터 토큰을 생성하고, 생성된 토큰을 파서로 넘겨주게 된다. MEGACO에 대한 Lexer class는 변환된 EBNF에서 토큰으로 사용하는 규칙들을 모아서 정의되었다. 그리고 렉서에서 토큰들 간의 모호성을 제거하기 위한 다른 방법들이 사용되었다.

3.2.1. SafeChar 토큰

SafeChar는 아래의 [표2]에서 나타난 것처럼 여러 개의 하위 토큰을 가지고 있는 토큰이다. 하지만 입력으로 'ALPHA'(알파벳 문자)가 주어졌을 경우 하위에 정의된 ALPHA 토큰과 모호성이 발견된다. 그리고 입력에 따라 다른 하위 토큰들과도 모호성이 발생된다. 따라서 ANTLR의 속성인 'protected' 키워드를 사용하여 새로운 SafeChar 토큰을 생성하였다. protected키워드를 사용한 토큰은 다른 토큰을 구성하기 위해서만 사용하겠다는 뜻이며, 이런 토큰은 파서 규칙에서 직접적으로 접근이 불가능하다.

```
SafeChar : DIGIT | ALPHA | '+' | '-' | '&' | '!' | '_' | '/'
         | '₩'' | '?' | '@' | '^' | '~' | '*' | '$' | '₩₩'
         | '(' | ')' | '%' | '|' | '.' ;
protected
ALPHA : ('a'..'z' | 'A'..'Z') ;
protected
DIGIT  : '0'..'9' ;
```

[표2] SafeChar의 정의

3.2.2. SafeChars 토큰

MEGACO 규칙에서는 메시지의 종류와 분류를 정의하기 위한 여러 가지 문자열 토큰이 존재한다. 하지만 이러한 문자열 토큰을 렉서에서 사용하면 SafeChar 토큰과 모호성이 발생하게 되므로 다른 방법을 사용하여 문자열 토큰을 정의하여야 한다. 먼저 문자열 토큰에 대한 정보를 문법 파일에서 'tokens' 부분에 정의하고, ANTLR의 옵션(Option) 중 'testLiterals'를 사용하여 접근하여야 한다. 이것은 그러한 문자열을 인식할 수 있는 토큰이 렉서에서 존재해야 사용할 수 있는데, 이를 위하여 SafeChars 토큰을 정의하였다.

SafeChars 토큰이 정의됨에 따라 파서 규칙에서 SafeChar의 하위 토큰은 직접적으로 접근이 불가능하게 되었다. 그러므로 원래 하위 토큰을 사용하는 부분은 SafeChars토큰을 인식하도록 수정되었으며, 약화된 규칙 사항은 3.3.4절에서 설명하는 Semantic Predicate를 사용하여 정당성을 검증받게 된다.

3.2.3. NonEscapeChar 토큰

NonEscapeChar는 원래 'x01'에서 'xFF'까지의 모든 문자에 해당하지만, MEGACO Lexer를 구현하기 위해서 여러 가지 다른 토큰을 선언해야 하기 때문에 하위의 토큰들과의 모호성을 없애 주어야 한다. 따라서 'x01'에서 'xFF' 사이의 모든 문자 중에서 MEGACO Lexer에서 사용하는 토큰들 이외의 문자들로 '~(NOT)' 연산자를 사용하여 새로 정의하였다. [표3]은 기존의 NonEscapeChar와 새로 정의된 NonEscapeChar를 보여준다.

	NonEscapeChar
원래 규칙	nonEscapeChar = ("₩}" / %x01-7C / %x7E-FF)
개정된 규칙	NonEscapeChar : ~('₩0'|'0'..'9'|'a'..'z'|'A'..'Z'|'+'|'-'|'&' |'!'|'_'|'/'|'₩''|'?'|'@'|'^'|'~'|'*'|'$'|'₩₩' |'('|')'|'%'|'|'|'.'| '|'₩t'|'₩r'|'₩n'|'"'|';' |'[' |']'|'{'|'}'|':'|','|'#'|'<'|'>'|'=') ;

[표3] 새로운 NonEscapeChar의 정의

개정된 NonEscapeChar 토큰은 원래 포함되어야 하는 다수의 문자들을 제외시키고 있다. 이것은 원래의 의미와 차이가 있으므로 파서 규칙에서는 제외된 많은 문자들을 다시 포함시켜야한다. 본 논문에서 개발한 MEGACO 파서에서는 이러한 문제를 [표4]에서 나타난 것과 같이 새로운 파서 규칙인 'nonEscapeChar'를 추가하여 해결하였다.

```
nonEscapeChar
  : "₩}" | NonEscapeChar | safeStrings | SEMI | LSBRKT
  | RSBRKT | LBRKT | COLON | COMMA | NEQUAL | L_T
  | G_T | EQUAL
  ;
```

[표4] nonEscapeChar 규칙의 정의

3.3. Parser class의 정의

파서는 Lexer로부터 받은 일련의 토큰을 이용하여, 입력이 파서에 정의된 규칙에 올바른지를 판단하는 구문 분석기 이다. ANTLR에서는 분석에 필요한 규칙들을 모두 Parser class에 정의하여야 한다. 규칙들 간의 비결정적인 부분은 ANTLR에서 제공하는 Syntactic predicate를 사용하여 해결하였고, 규칙의 정당성을 보다 상세하게 검사하기 위해서 Semantic predicate를 사용하였다.

3.3.1. safeStrings 규칙

ANTLR에서는 SafeChars를 인식하는 파서 규칙에서 'tokens' 부분에 존재하는 문자열이 입력으로 주어질 경우 SafeChars 토큰으로 인식하는 것이 아니라 문자열 토큰으로 인식하게 된다. 따라서 정당한 SafeChars 토큰을 인식하기 위해서 safeString 규칙을 정의하였다. 그리고 이러한 경우에는 하나의 문자열이 여러 개의 토큰으로 분리되기 때문에, 여러 개를 하나로 통합하여야 한다. 이를 위해 [표5]와 같이 safeStrings 규칙을 정의하였다.

```
safeStrings
    :   safeString safeString_list   ;
safeString_list
    :   (safeString) => safeString safeString_list
    |   ;
safeString
    :   SafeChars
    |   addToken
    |   auditCapToken
        ..... ;
```

[표5] safeString과 safeStrings

3.3.2. domainAddress와 digitMap 규칙

domainAddress와 digitMap 규칙은 자체적으로 비결정적인 부분이 존재한다. 하지만 ANTLR에서는 이러한 부분을 해결할 수 있는 방법이 존재하지 않는다. 따라서 domainAddress와 digitMap 규칙은 의미적으로 같으면서도 표현방식이 틀린 다른 형태의 규칙으로써 새로 정의하였다. [표6]은 새로 정의된 domainAddress를 보여준다.

```
domainAddress
  : LSBRKT safeStrings RSBRKT { sm.isIPAddress() }?
  ;
```

[표6] 새로 정의된 domainAddress

[표6]은 domainAddress를 safeStrings 규칙으로 인식한 다음, ANTLR에서 제공하는 Semantic predicate를 사용하여 인식된 safeStrings 규칙이 실제로 domainAddress에 맞는지 정당성을 검사하는 부분이 포함되어 있다.

3.3.3. Syntactic predicate

ANTLR의 Syntactic predicate는 비결정적인 두 개 이상의 규칙이 있을 때, Lookahead를 사용하여 결정적인 규칙으로 바꿔주는 역할을 한다. [표7]은 MEGACO 파서에서 사용된 Syntactic predicate의 예를 나타내고 있다.

```
auditReturnParameter
  :  (modemToken) => modemDescriptor
  | (muxToken) => muxDescriptor
  | auditItem
   ..... ;

auditItem
  :  muxToken
  | modemToken
  | mediaToken
   ..... ;
```

[표7] Syntactic predicate의 예

3.3.4. Semantic predicate

ANTLR의 Semantic predicate는 메소드(Method)를 사용하여 인식된 규칙이 정당한 것인지 판단하는 기능을 수행한다. 예를 들면, 아래 [표8]에서처럼 1~4자리의 정수를 인식하는 errorCode 규칙은 SafeChars라는 토큰을 인식하고, 인식된 Safechars 토큰이 1~4자리 정수인지를 판단하기 위해서 Semantic predicate가 사용된다.

```
{
  bool isDigitRange(const char *s, int start, int end) {
      int length = strlen(s);
      if(length < start || length > end)
          return false;
      for (int i = 0; s[i] != '₩0'; i++) {
          if ( s[i] < '0' || s[i] > '9')
            return false;
          }
          return true;
      }
}
.....
errorCode
    :   ec:SafeChars
        { isDigitRange(ec->getText().data(), 1, 4) }?
    ;
```

[표8] Semantic predicate의 예

4. 파스 트리(Parse Tree)

파스 트리는 입력된 메시지를 파싱한 결과를 트리로 저장한다. 이것은 이진 트리 형식으로 구성되고, 각 규칙에 매개변수를 전달할 수 있는 ANTLR의 기능을 사용하였다.

4.1. 파서 규칙에서 매개변수의 전달

ANTLR에서 각 규칙에 대하여 매개변수를 설정할 수 있는 기능이다. 규칙이 생성되는 파서에서 각각의 메서드로 존재하기 때문에 사용가능 하다. 그리고 매개변수뿐만 아니라 반환 형식(Return type)을 명시할 수도 있다. [표9]는 MEGACO 트리를 구성하기 위해 매개변수를 사용한 예를 보여준다.

```
message
  { mt_node *msgNode = buildNode(ntt.message); }
  :  SafeChars mId[msgNode] messageBody[msgNode]
  ;
messageBody[mt_node *parent]
  { mt_node *msgBody = buildNode(ntt.messageBody); }
  : ( errorDescriptor | transactionList )
  { addNode(parent, msgBody); }
  ;
```

[표9] 매개변수를 사용한 파스 트리의 구성

4.2. 트리 정의 및 생성

위에서 설명한 매개변수를 정의하는 방법을 사용하여 MEGACO 메시지에 대한 파스 트리를 생성할 수 있다. 기본적인 원리는 위의 [표9]와 같이 상위 규칙에서 자신의 노드를 생성하여 매개변수로 자신의 하위 규칙으로 전달하고, 그 하위 규칙에서는 전달받은 노드에 자신의 노드를 생성하여 자식 노드로 추가하는 것이다.

5. 결 론

본 논문에서는 여러 응용분야에서 널리 사용되고 있는 ANTLR 파서 생성기를 사용하여 MEGACO 파서를 생성하였다. MEGACO 명세에 정의된 ABNF문법에는 많은 수의 비결정적인 규칙과 모호한 토큰이 존재하므로 이 문법으로부터 직접적으로 파서를 개발하기에는 적절하지 않다. 따라서 다양한 ANTLR의 기능을 사용하여 기존의 비결정적인 요소와 모호한 부분을 제거하고 파서를 생성하였다.

본 논문에서 사용한 비결정적인 구분을 제거하는 방법은 다른 여러 가지 명세서에서 제공하는 ABNF 명세를 ANTLR을 사용하여 파서를 개발할 때 유용하게 사용될 것으로 기대된다.

6. 참고문헌

[1] Jonathan Davidson, et al, "Voice over IP Fundamentals," Cisco Press, March 2000, 373 pages

[2] C. Groves, M. Pantaleo, et al, "The Megaco/H.248 Gateway Control Protocol, version 2", IETF internet draft, April 2003

[3] M. E. Lesk, E. SchmidtLex, "A Lexical Analyzer Generator"

[4] Stephen C. Johnson, "Yacc: Yet Another Compiler-Compiler", AT&T Bell Laboratories.

[5] GNU Flex, "http://www.gnu.org/software/flex/flex.html"

[6] GNU Bison, "http://www.gnu.org/software/bison/bison.html"

[7] Terence Parr, Russell Quong, "ANTLR: A Predicated-LL(k) Parser Generator", Fri Jun 13, 2003.