JavaCC를 이용하여 당신만의 언어를 만들자.

중앙대학교 컴퓨터공학과
자바 동호회
JSTORM
http://www.jstorm.pe.kr

Document Information

Document title:	JavaCC를 이용하여 당신만의 언어를 만들자.
Document file name:	javaCC_junoyoon_jstorm_final.doc
Revision number:	<1.0>
Issued by:	<윤준호> (junoyoon@selab.snu.ac.kr)
Issue Date:	<2002/9/01 >
Status:	Final

Content Information

Audience	중급, 고급
Abstract	JavaCC를 이용하여 계산기 프로그램을 만들어 봄으로써 Parser를 만들고 사용하는 방법을 배운다.
Reference	http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-cooltools.html 번역
Benchmark information	

Table of Contents

JavaCC

자바 컴파일러가 어떻게 돌아가는지 궁금하지 않는가? 당신이 직접 만든 마크업 문서(HTML/XML 같은 Standard 가 아닌)를 파싱할 수 있는 파서가 필요한가? 당신만의 언어를 만들고 싶은가? JavaCC 는 위에 말한 모든 것를 자바로 할 수 있게 해준다. 컴파일러와 인터프린터가 어떻게 작동하는지 알고 싶거나 자바를 계승하는 새로운 언어를 만들고 싶다는 굳건한 의지가 있다면 이 글을 읽어보도록 하라. 여기서는 작은 커멘드 라인 계산기를 만들어 봄으로써 당신만의 언어를 만들어 보도록 하겠다.

컴파일러 구축의 기초

프로그래밍 언어는 컴파일러와 인터프리터 언어로 구분되어 진다. 물론 그 경계가 가면 갈수록 모호해 지긴 하지만 서리.. 그러나 걱정하지 마라. 여기서 알아볼 것은 컴파일러를 만들건 인터프리터 언어를 만들건 상관없이 적용된다. 이 글에서는 "컴파일러"라는 용어를 사용할 것이다. 그러나 이 글에서 컴파일러는 "인터프리터"라는 뜻도 포함하고 있다.

컴파일러는 소스코드가 주어지면 다음과 같은 3 개의 주요 작업을 실시한다.

1. Lexical analysis
2. Syntactic analysis
3. Code generation or execution

대부분의 컴파일러의 작업은 1 번과 2 번에 집중에 되어 있다. Lexical Analysis 와 Syntatic Analysis 는 프로그램 소스코드를 이해하고 그 의미가 정확한지 보장하는 역할을 맡는다. 우리는 이러한 작업을 파싱이라 부린다. 또한 파싱을 하는 프로그램을 바로 파서라고 한다.

Lexical analysis (lexing)

Lexical analysis 는 먼저 프로그램을 대강 살펴본 다음 프로그램을 토큰으로 나누는 역할을 한다. 토큰 이라는 것은 소스 코드의 중요한 부분들을 의미한다. 예를 들어 키워드나, 마침표, 문자들(예 숫자) 또는 스트링이 토큰이라고 할 수 있다. 토큰이 아닌 것에는 공백(스페이스 같은 거), 구분자 또는 주석 같은 것을 들 수 있겠다.

Syntactic analysis (parsing)

syntactic analysis 단계에서 파서는 프로그램 소스 코드에서 소스코드가 뭘 의미하는 지를 뽑아낸다. 이 과정에는 프로그램의 문법이 틀리지나 않았는지 검사하고 또 프로그램를 컴파일러가 이해할 수 있는 형태로 만드는 작업이 포함된다.

컴퓨터 언어론에서는 프로그램, 문법, 언어 이 3 개의 요소들에 대해서 설명하고 있다. 이를 정의하자면 프로그램은 기냥 토큰들을 열거한 것이다. 문법은 옳바른 의미적으로 정확한 프로그램을 만들기 위한 룰이다. 문법에서 제시하는 규칙을 그대로 따르고 있는 프로그램만이 correct 하다고 할 수 있다. 언어는 모든 문법 규칙을 만족하는 프로그램들을 의미한다.

syntactic analysis 동안 컴파일러는 언어의 문법에 정의되어 있는 규칙을 이용하여 소스코드를 관찰한다. 소스의 문법이 틀렸다면 컴파일러는 바로 에러 메시지를 출력한다. 프로그램을 관찰 하는 동시에 컴파일러는 컴퓨터 프로그램을 컴퓨터가 쉽게 처리할 수 있는 형태로 바꿔놓는다.

언어의 문법 규칙은 EBNF(Extended Backus-Naur-Form) 표기기법을 이용하여 모호함이 없이 기술될 수 있다. (EBNF 에 대해 더 알고 싶다면 Resources 를 살펴보라). EBNF 는 production rule 이라는 관점에서 문법을 작성한다. production rule 은 문법 요소(예를 들어 더 이상 줄일 수 없는 문자들 또는 복합 요소들)들은 다른 문법 요소들로 이루어 질 수 있다는 것을 의미한다.

더 이상 줄일 수 없는 문자들에는 키워드 또는 쉼표 같은 것이 있다. 복합 요소는 production rule 을 적용하여 유도되어 진다. Production rule 는 다음과 같은 형식을 따른다.

```
GRAMMAR_ELEMENT := list of grammar elements
                |  alternate list of grammar elements
```

예로써 수학적 계산을 표현하는 조그마한 언어가 있다고 치고, 그 문법을 알아보도록 하자.

```
expr    :=    number
      |     expr '+' expr
      |     expr '-' expr
      |     expr '*' expr
      |     expr '/' expr
      |     '(' expr ')'
      |    - expr
number    :=    digit+ ('.' digit+)?
digit     :=    '0' | '1' | '2' | '3' | '4' | '5' | '6' |
                '7' | '8' | '9'
```

3 개의 production rule 이 문법 요소들를 정의하고 있다.

- expr
- number
- digit

이러한 문법을 가진 언어를 이용하여 계산식을 표현할 수 있다. expr 은 다시 2 개의 expr 사이에 연산기호를 끼워 넣은 것이 될 수도 있고, 가로 안에 있는 expr 이 될 수도 있고 음수 expr 이 될 수도 있다. Number 는 부동소수점 숫자이다. 여기서 digit 는 0 ~ 9 까지의 숫자를 의미하고 있다.

Code generation or execution

파서가 프로그램을 에어 없이 잘 파싱 했다면 컴파일러가 처리하기 쉬운 형태로 프로그램이 변경되었을 것이다. 이 형태로부터 (1)기계어 코드(자바에서는 자바 바이트 코드)를 만들거나 (2)바로 실행해 버리는 건 쉬운 일이다. (1)이 바로 컴파일 이고, (2)가 인터프리팅이다.

JavaCC

JavaCC 는 무료로 사용할 수 있는 파서 생성기이다. 이것은 프로그래밍 언어의 문법을 정의하는 작업을 위해, 자바 언어를 확장한 거라고 보면 맞을 것이다. JavaCC 는 원래 Sun 에 의해서 만들어졌지만 지금은 MetaMata 라는 회사가 개발하고 있다. 요즘은 이게 아무래도 Webgain 이라는 회사로 또 넘어간 듯하다.

먼저 EBNF 와 비슷한 형태로 문법을 만들고 이를 JavaCC 포멧으로 변경하여 이를 JavaCC 에 입력하면 JavaCC 가 알아서 파서를 만들어 주게 된다. JavaCC 는 자바진영에서 가장 유명한 파서 생성기이구, 이미 JavaCC 를 위한 많은 문법들을 제공되어져 있어서 사용하기 편리하다.

간단한 계산기 개발하기

이제부터 여기서는 앞에서 만들었던 계산 언어와 JavaCC 를 이용하여 간단한 커멘드 라인 계산기를 만들어 볼 것이다. 먼저 EBNF 문법을 JavaCC 포멧으로 바꾸고 그것을 Arithmetic.jj 라는 파일로 저장한다.

```
options
{
    LOOKAHEAD=2;
}

PARSER_BEGIN(Arithmetic)

public class Arithmetic
{
}

PARSER_END(Arithmetic)

SKIP :
{
    " "
|   "\r"
|   "\t"
}
```

```
TOKEN:
{
    < NUMBER: (<DIGIT>)+ ( "." (<DIGIT>)+ )? >
|   < DIGIT: ["0"-"9"] >
}

double expr():
{
}
{
    term() ( "+" expr() | "-" expr() )*
}

double term():
{
}
{
    unary() ( "*" term() | "/" term() )*
}

double unary():
{
}
{
    "-" element() | element()
}

double element():
{
}
{
    <NUMBER> | "(" expr() ")"
}
```

위의 코드를 살펴본다면 JavaCC 에 입력될 문법을 어떻게 정의하는지 알 수 있을 것이다. 코드내의 Option 부분은 문법상의 몇 가지 선택 사항들을 정의하는 부분이다. 여기서는 lookahead 를 2 로 설정했다. 다른 Option 들은 javaCC 의 디버깅 기능을 제어하는데 쓰인다. 이러한 옵션은 JavaCC 커멘드 라인 상에서 직접 설정해 줄 수도 있다.

PARSER_BEGIN 구문은 뒤이어 파서 클래스 정의가 계속된다는 것을 나타내 준다. JavaCC 는 각각의 파서를 위해 하나의 자바 클래스를 생성한다. 여기서는 Parser 클래스를 Arithmetic 이라고 이름 지었다. 지금은 단지 빈 클래스 정의만이 필요하다. JavaCC 에서는 파싱과 관련된 여러 선언들은 뒤쪽 추가하도록 되어있다. 파서 클래스 정의는 PARSER_END 구문으로 끝나게 된다.

```
PARSER_BEGIN(Arithmetic)

public class Arithmetic
{
}

PARSER_END(Arithmetic)
```

SKIP 부분은 파싱할 때 그냥 무시해도 될만한 문자들을 기록하는 부분이다. 여기서는 공백 문자들이 SKIP 에 정의되어 있다. 다음에는 TOKEN 부분에 만들 언어의 토큰들을 정의해 넣어야 한다. 여기서는 숫자들을 토큰으로 정의했다. 여기서 한가지만 주의하자. JavaCC 에서는 EBNF 와는 틀리게 토큰의 정의와 production rule 의 정의를 서로 구분해 놓고 있다. SKIP 과 TOKEN 부분에서는 Lexical Analysis 만을 담당하고 있다.

```
SKIP :
{
    " "
|   "\r"
|   "\t"
}

TOKEN:
{
    < NUMBER: (<DIGIT>)+ ( "." (<DIGIT>)+ )? >
|   < DIGIT: ["0"-"9"] >
}
```

다음으로 top-level 의 문법 요소인 expr 의 production rule 을 정의할 차례이다. JavaCC 에서 expr 을 정의할 때 EBNF 에서의 할 때와 어떻게 다른지 주의 깊게 살펴보기 바란다. 사실상 EBNF 정의는 모호한 구석이 있었다. 즉 한 개의 expression 에 대해서 EBNF 로 정의를 하게 되면 다중의 표현을 할 수도 있다는 것이다. 예를 들어 1+2*3 이라는 expression 을 살펴보자. 그림 1 과 같이 1+2 을 expr 로 정의하고 이후에 expr*3 으로 표현될 수 있다.

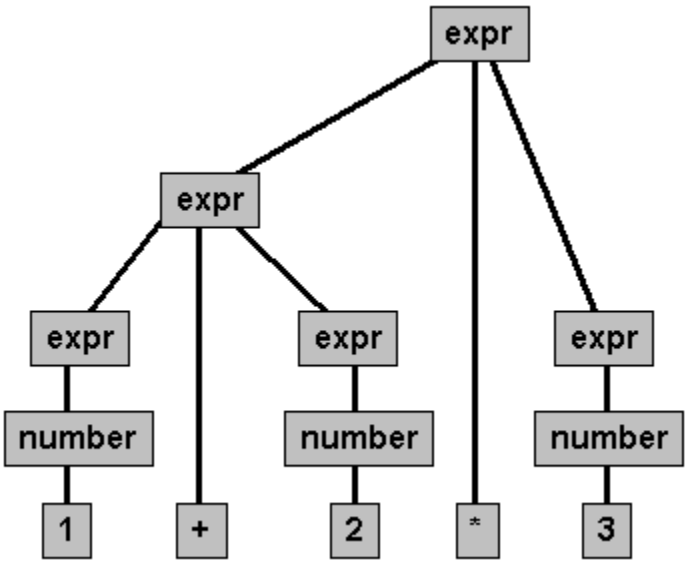

Figure 1. EBNF parse tree of 1+2*3

또 그림 2 와 같이 먼저 2*3 을 expr 로 정의하고 이후에 1+expr 로 정의될 수도 있다.

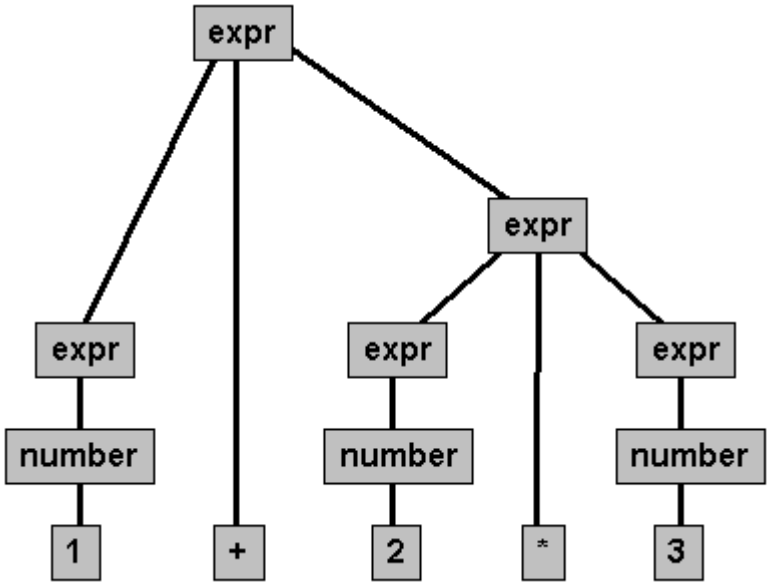

Figure 2. Alternative EBNF parse tree of 1+2*3

JavaCC 를 이용할 때는 반드시 문법을 모호하지 않는 상태로 만들어야 한다. 결과적으로 expr 을 expr, term, unary, element 총 4 개의 production rule 로 나눠야만 한다. 자 이제 1+2*3 이 어떻게 파싱되는지 보자.

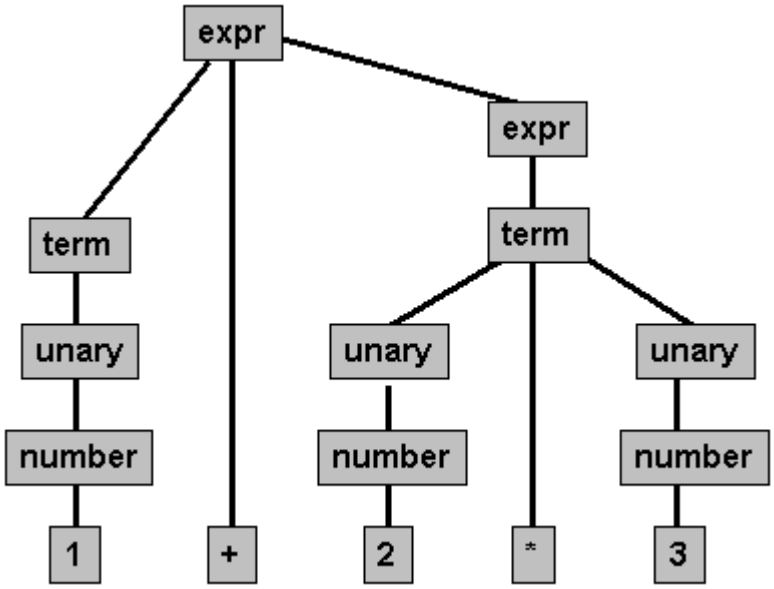

Figure 3. Parse tree of 1+2*3

```
double expr():
{
}
{
    term() ( "+" expr() | "-" expr() )*
}
```

```
double term():
{
}
{
    unary() ( "*" term() | "/" term() )*
}

double unary():
{
}
{
    "-" element() | element()
}

double element():
{
}
{
    <NUMBER> | "(" expr() ")"
}
```

커멘드 라인에서 JavaCC 를 실행함으로써 지금 만든 문법을 체크할 수 있다.

```
javacc Arithmetic.jj
Java Compiler Compiler Version 1.1 (Parser Generator)
Copyright (c) 1996-1999 Sun Microsystems, Inc.
Copyright (c) 1997-1999 Metamata, Inc.
(type "javacc" with no arguments for help)
Reading from file Arithmetic.jj . . .
Warning: Lookahead adequacy checking not being performed since
option LOOKAHEAD
is more than 1.  Set option FORCE_LA_CHECK to true to force
checking.
Parser generated with 0 errors and 1 warnings.
```

JavaCC 를 실행시키면 먼저 문법에 별 문제가 없는지 검사하고 나서, 최종적으로 여러 개의 Java 소스 파일들을 만들어 낸다.

```
TokenMgrError.java
ParseException.java
Token.java
ASCII_CharStream.java
Arithmetic.java
ArithmeticConstants.java
ArithmeticTokenManager.java
```

위의 파일들을 모두 사용하여 파서가 구성되는 것이다. Arithmetic 객체를 생성함으로써 파서를 실행시킬 수 있다.

```
public class Arithmetic implements ArithmeticConstants
{
   public Arithmetic(java.io.InputStream stream) { ... }
    public Arithmetic(java.io.Reader stream) { ... }
```

```
   public Arithmetic(ArithmeticTokenManager tm) { ... }
   static final public double expr() throws ParseException
     { ... }
   static final public double term() throws ParseException
     { ... }
   static final public double unary() throws ParseException
     { ... }
   static final public double element() throws ParseException
     { ... }

   static public void ReInit(java.io.InputStream stream)
{ ... }
   static public void ReInit(java.io.Reader stream) { ... }
   public void ReInit(ArithmeticTokenManager tm) { ... }

   static final public Token getNextToken() { ... }
   static final public Token getToken(int index) { ... }

   static final public ParseException generateParseException()
     { ... }

   static final public void enable_tracing() { ... }
   static final public void disable_tracing() { ... }
}
```

이 파서를 사용할때는, 여러 생성자중에 하나를 사용하여 이 클래스를 인스턴스로 만들어야 한다. 생성자에서는 InputStream 이나 Reader 또는 ArithmeticTokenManager 를 파라메터로 전달받는다. 이 파라메터를 이용하여 파싱하기 원하는 문장이나 스트림을 넣어줄 수 있다.

```
Arithmetic parser = new Arithmetic(System.in);
parser.expr();
```

그러나 Arithmetic.jj 에서 단지 문법만 정의를 했기 때문에 위처럼 하더라도 아무것도 실행이 않될 것이다. 아직 여기서는 실제 계산을 수행하는 코드를 추가하지 않았음을 명심하자. 이걸 하기 위해서는 문법에 적절한 action 을 추가해 줘야 한다. 첨부로 주어진 Calcualtor.jj 에는 완전한 계산기 기능을 포함하고 있다.

```
options
{
    LOOKAHEAD=2;
}

PARSER_BEGIN(Calculator)

public class Calculator
{
    public static void main(String args[]) throws
ParseException
    {
        Calculator parser = new Calculator(System.in);
        while (true)
        {
            parser.parseOneLine();
        }
```

```
    }
}

PARSER_END(Calculator)

SKIP :
{
    " "
|   "\r"
|   "\t"
}

TOKEN:
{
    < NUMBER: (<DIGIT>)+ ( "." (<DIGIT>)+ )? >
|   < DIGIT: ["0"-"9"] >
|   < EOL: "\n" >
}

void parseOneLine():
{
    double a;
}
{
    a=expr() <EOL>      { System.out.println(a); }
  | <EOL>
  | <EOF>               { System.exit(-1); }
}

double expr():
{
    double a;
    double b;
}
{
    a=term()
    (
        "+" b=expr()    { a += b; }
    |   "-" b=expr()    { a -= b; }
    )*
                        { return a; }
}

double term():
{
    double a;
    double b;
}
{
    a=unary()
    (
        "*" b=term()    { a *= b; }
    |   "/" b=term()    { a /= b; }
    )*
                        { return a; }
```

```
}

double unary():
{
    double a;
}
{
    "-" a=element()      { return -a; }
|   a=element()          { return a; }
}

double element():
{
    Token t;
    double a;
}
{
    t=<NUMBER>
                    { return
Double.parseDouble(t.toString()); }
|   "(" a=expr() ")"     { return a; }
}
```

메인 메소드에서는 먼저 parser 객체를 인스턴스화한다. 이 객체는 standard input 에서 입력을 받고 무한 루프에서 parseOneLine() 메소드를 호출한다. parseOneLine() 메소드는 문법에 추가되어져 있다. 이 메소드 내에서는 간단히 라인 별로 입력한 표현들을 가져오는 역할을 한다. 만약 그냥 빈 문장을 쳐 넣더라도 상관없고, 만약 end of the file 을 만나게 된다면 프로그램을 종료시킨다.

```
void parseOneLine():
{
    double a;
}
{
    a=expr() <EOL>       { System.out.println(a); }
  | <EOL>
  | <EOF>                { System.exit(-1); }
}
```

또 여기서는 원래 문법에서의 리턴타입 돌을 double 을 리턴하도록 변경하였다. 파싱을 하고 그 결과를 상위 트리로 올려줘야할 필요가 있을때마다 적절한 계산을 수행하고 있다. 덧붙여서 문법 요소들의 결과들을 저장할 수 있도록 local 변수를 만들어 줬다. 에를 들어 a=element()는 element 를 파싱하고 그 결과를 변수 a 에 저장한다. 이런식으로 모두 좌측에 파싱된 요소들의 결과값을 저장한다. 저장되면 우측의 Action 코드에 저장된 변수값이 전달되고 적절한 계산이 수행되는 것이다.Actions 은 관련된 문법이 인풋 스트임과 매칭될 때 실행되는 자바 코드를 말한다.

JavaCC 를 이용하여 얼마나 쉽게 계산기가 만들어 졌는지 주의깊게 보기 바란다. 여기에 더 기능을 추가하는 것도 쉽다.

결론

JavaCC, 자바 파서 생성기, 는 프로그래밍 언어를 만드는 것을 아주 쉽게 해준다. JavaCC 에서는 문법을 정의하고, 관련된 Action 을 만들기 위한 high-level 노테이션들을 제공하고 있다. 또한 직접 하나 하나 프로그래밍 하는 것보다 훠얼씬 소스 코드 읽기도 쉽다. Java 1.2 에 대한 문법도 공개가 되어 있으니 다운받아서 JavaCC 에 돌려볼 수 있다. 이를 이용해서 당신만의 javadoc 이나 java 인터프리터, java 소스 출력기등을 만들 수 있다. 사실상 javaCC 를 이용하여 뭐든지 할 수 있을 것이다.

JavaCC 2 개의 유틸리티 툴을 가지고 있다. JJDoc 은 자동으로 javadoc 과 비슷한 형식으로 HTML 형태의 문법 도큐먼트를 만들어 준다. JJTree 는 프로그램을 파싱할 때 자동으로 트리 구조를 만드는 액션을 생성해낸다.

End of Document