JLex Manual

Elliot Berk

Department of Computer Science, Princeton University

도운이

박종수

clhitter (at) bawi.org

JLex : Java용 Lexical analyzer generator 최신 버젼을
http://www.cs.princeton.edu/~appel/modern/java/JLex
(http://www.cs.princeton.edu/~appel/modern/java/JLex/)에서 구할 수 있다.

고친 과정

고침 번역 1.0	2001년 3월 20일 고친이 박종수
고침 Version1.2	1997년 5월 5일
고침 Manual revision	1997년 10월 29일
고침 Last updated for JLex 1.2.5	2000년 9월 6일

1. 소개

Lexical analyzer는 문자열 입력 스트림을 토큰으로 분해하는 프로그램이다. 어휘 분석기를 수동으로 작성하는 것은 난해한 작업이므로, 좀 더 쉽게 할 수 있는 도구들이 개발되었다.

이중 가장 잘 알려진 것이 Lex이다. Lex는 C 언어를 타겟으로 하는 UNIX 용 lexical analyzer generator이다. Lex는 lexical analyzer의 세부 사항들을 담고 있는 특정한 형태의 specification 파일을 분석하여, table-driven 방식의 lexical analyzer C 소스 코드를 생성한다.

JLex는 Lex의 lexical analyzer 생성 모델을 기반으로 한다. JLex는 Lex와 비슷한 specification 파일을 받아들여서 해당 기능을 수행하는 Java 소스 코드를 생성한다.

2. JLex specifications

JLex specification은 %%로 구분된 세 section으로 구성된다. 형식은 다음과 같다

```
사용자 코드
%%
JLex 지시문
%%
정규식 rules
```

%%는 각 section을 구분하는 역할을 하며, 해당 라인의 제일 처음에 위치해야만 한다. 이외의 위치에 있는 %%는 무시되며 추가의 선언문이나 코드들의 명확한 해석을 위해 사용을 자제해야 한다.

specification 파일의 첫 section인 사용자 코드 section은 출력 파일에 그대로 복사되는 부분이다. 이 영역은 유틸리티 클래스나 리턴 type을 기술할 수 있도록 제공되는 공간이다.

두 번 째인 JLex 지시문 section에서는, 매크로가 정의되고 상태 이름이 선언된다.

세 번째 section에서는 lexical analyzer의 rule이 정의된다. : 각 rule은 option 상태 list, 정규식, action의 세 부분으로 구성된다.

2.1. User Code

사용자 코드는 첫 번째 %%의 앞부분에 해당되며 이곳의 내용은, JLex의 출력물인 lexical analyzer 소스코드의 처음에 그대로 복사된다. 따라서, lexer 소스가 package 선언이나 중요 외부 클래스를 필요로 하는 경우, 사용자 코드 section 첫 부분에 선언할 수 있다.

2.2. JLex 지시문

JLex 지시문 section은 첫 번째 %%에서부터 두 번째 %%까지 계속된다. 각 JLex 지시문은 줄의 처음부터 시작되어 그 줄에서 안에서 끝나야 한다.

2.2.1. Lexical analyzer 클래스의 내부 코드

%{...%} 지시문안에는 lexical analyzer 클래스에 복사될 자바 코드를 작성할 수 있다. 형식은 다음과 같다.

```
%{
<code>
%}
```

%{와 %}는 각 줄의 처음에 위치해야만 한다. <code>에 있는 자바 코드는 다음과 같은 형태로 lexical analyzer 클래스에 복사된다.

```
class Yylex {
... <code> ...
```

```
}
```

위와 같이하여 생성될 lexical analyzer 클래스에 들어갈 변수와 함수들을 정의할 수 있다. yy로 시작되는 변수 이름은 lexical analyzer class들을 위하여 예약되어 있기 때문에 피해야 한다.

2.2.2. Lexical analyzer 클래스의 초기화 코드

%init{ ... %init} 지시문안에는 lexical analyzer 클래스의 생성자에 복사될 자바 코드를 작성할 수 있다.

```
%init{
<code>
%init}
```

%init{와 %init} 지시문들은 줄의 처음에 위치해야만 한다. <code>에 있는 자바 코드는 다음과 같은 형태로 lexical analyzer 클래스 생성자에 복사된다.

```
class YYlex {
Yylex( ) {
... <code> ...
}
}
```

위와 같이하여 lexical analyzer 클래스의 생성자 코드를 작성할 수 있다. yy로 시작되는 변수 이름은 lexical analyzer class들을 위하여 예약되어 있기 때문에 피해야 한다.

%init{ ... %init} 지시문안에서는 exception을 throw하거나 다른 함수로부터의 것을 전파시킬 수 있다. 이들 exception을 선언하려면 %initthrow{ ... %initthrow} 지시문을 사용하면 된다.

```
%initthrow{
<exception[1]>[,<exception[1]>,...]
%initthrow}
```

이렇게 선언된 exception들은 lexical analyzer 생성자에 다음과 같은 형태로 복사된다.

```
Yylex()
throws <exception[1]>[,<exception[1]>,...]
{
... <code> ...
}
```

%init{ ... %init} 지시문안의 자바 코드에서 선언되지 않은 exception을 throw하는 경우 생성된 lexical analyzer 소스 파일이 성공적으로 컴파일 되지 못할 것이다.

2.2.3. Lexical analyzer 클래스의 End-of-File 코드

%eof{ ... %eof} 지시문안에는 lexical analyzer 클래스가 end-of-file을 만난 이후에 수행할 자바 코드를 작성할 수 있다.

```
%eof{
<code>
%eof}
```

%eof{와 %eof} 지시문들은 각 줄의 처음에 위치해야만 한다. <code>부분에 있는 자바 코드는 lexical analyzer가 처리하는 입력 파일의 end-of-file에 도달된 직후에 최대 1번까지 수행된다.

%eof{ ... %eof} 지시문안의 code들은 exception을 throw하거나 다른 함수로부터의 것을 전파시킬 수 있다. 이들 exception을 선언하려면 %eofthrow{ ... %eofthrow} 지시문을 사용하면 된다.

```
%eofthrow{
<exception[1]>[,<exception[1]>,...]
%eofthrow}
```

이렇게 선언된 exception들은 lexical analyzer안의 end-of-file 도달 후 호출되는 함수 선언 부에 다음과 같은 형태로 복사된다.

```
private void yy_do_eof()
throws <exception[1]>[,<exception[1]>,...]
{
... <code> ...
}
```

함수의 몸체 부에 있는 <code>는 %eof{ ... %eof} 지시문안의 <code>로부터 복사된다. 만약 여기서 %eofthrow{ ... %eofthrow} 지시문에서 선언하지 않은 exception을 throw하는 경우 생성된 lexical analyzer 소스 파일이 성공적으로 컴파일 되지 못할 것이다.

2.2.4. 매크로 정의

매크로 정의는 specification 파일의 JLex 지시문 section에서 이루어진다. 각각의 매크로 정의문은 매크로 이름, =, 정의부가 들어간 하나의 줄로 작성된다. 다음과 같이 형식을 요약할 수 있다.

```
<name>=<definition>
```

빈칸이나 탭과 같은 개행 문자가 아닌 공백들은, 매크로 이름과 등호 사이나 등호와 정의부 사이에 선택적으로 삽입될 수 있다. 각 매크로 정의문은 한 줄 안에서 끝나야만 한다.

매크로 이름은 알파벳이나 밑줄로 시작하여 알파벳, 아라비아 숫자, 밑줄의 나열이 따라오는 형태의 적법한 identifier여야 한다.

매크로 정의문은 적법한 정규식이어야 한다. 자세한 내용은 아래 section을 참고하라.

매크로 정의문은 정규식 중간에 {<name>}과 같은 형식으로 다른 매크로 정의가 포함되는 것을 허용한다. 하지만, 함수나 nonterminal과 같이 상호 재귀적으로 호출되는 것을 허용하지 않는 '매크로'라는 것을 명심해야 한다. 매크로 정의문에서 발생하는 순환 문제는 예상치 못한 결과를 발생시킬 수 있다.

2.2.5. 상태 선언

Lexical 상태들은 주어진 입력에 어떤 정규식을 적용해야 할지를 판단할 때 사용된다. 다음과 같은 형태로 JLex 지시문안에 선언된다.

```
%상태 state[0][,state[1],state[2],...]
```

각 lexical 상태 선언문은 한 줄 안에 포함되어야 한다. 여러 선언문들이 같은 JLex specification에 포함될 수 있기 때문에, 여러 상태 선언문들이 여러 줄로 쪼개어져 작성될 수 있다.

상태명은 알파벳이나 밑줄로 시작하여 알파벳, 아라비아 숫자, 밑줄의 나열이 따라오는 형태의 적법한 identifier여야 한다.

JLex에 의해 하나의 lexical 상태가 암시적으로 선언된다. 생성된 lexical analyzer가 시작될 때 YYINITIAL이라 불리는 상태를 갖는다.

lexical analyze의 규칙들은 optional 상태 리스트로 시작된다. 상태 list가 주어진 경우 lexical analyzer가 명시된 상태 중의 하나일 경우에만 lexical 규칙이 적용된다. 상태 list가 없는 경우 lexical 규칙은 현재의 lexical analyzer의 상태에 무관하게 적용된다.

JLex specification이 상태도 선언하지 않고 lexical 규칙 앞에 상태 list를 두지도 않는 경우, 생성되는 lexer의 상태는 YYINITIAL로 계속 유지된다. Lexical 규칙 앞에 상태 list가 없기 때문에, 암시적으로 선언되는 YYINITAIL을 포함한 모든 상태에 대해 규칙이 적용된다. 따라서, specification에서 상태가 전혀 사용되지 않은 경우라도 예상치 못한 결과가 발생하진 않는다.

State들은 생성될 lexical analyzer 클래스안에서 상수로 선언된다. 선언된 상태에 대응되는 상수는 상태명과 같은 이름을 갖는다. 따라서, 규칙의 action 부분에 선언된 변수명이 상태명과 중첩되는 것을 조심해야 한다. 관례적으로 상태명을 전부 대문자로 하여 상수라는 것을 상기시키게끔 한다.

2.2.6. 문자수 세기 기능

문자수 세기 기능은 default로 꺼져있지만, %char 지시문를 사용하여 동작시킬 수 있다.

```
%char
```

정규식과 일치된 영역의 Zero-based 문자 번호가 yychar라는 이름의 integer 변수에 기록된다.

2.2.7. 줄번호 세기 기능

줄번호 세기는 default로 꺼져있지만, %line 지시문를 사용하여 동작시킬 수 있다.

```
%line
```

정규식과 일치된 영역의 Zero-based 줄번호가 yyline라는 이름의 integer 변수에 기록된다.

2.2.8. Java CUP과의 호환성

Java CUP은 Georgia Tech University의 Scott Hudson이 처음 만들기 시작하여, Frank Flannery, Dan Wang, C.Scott Ananian에 의해 관리, 확장된 parser generator이다. 자세한 설명은 다음 URL에서 찾아볼 수 있다. http://www.cs.princeton.edu/~appel/modern/java/CUP/. Java CUP과의 호환 기능은 default로 꺼져있지만, 다음과 같은 JLex 지시문를 사용하여 동작시킬 수 있다.

```
%cup
```

이렇게 하면 java_cup.runtime.Scanner interface에서 정의된 규칙에 따르도록 scanner가 생성된다. 다음과 같은 지시문들로 똑같은 결과를 얻을 수 있다.

```
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
```

다음 section에서 이들 지시문에 대한 보다 자세한 설명을 얻을 수 있으며, CUP 매뉴얼에서 CUP과 JLex를 같이 사용하는 방법에 대한 보다 자세한 설명을 얻을 수 있을 것이다.

2.2.9. Lexical Analyzer 구성요소 이름 변경법

여기서 설명할 지시문들을 사용하여 생성될 lexical analyzer 클래스, 토큰화 함수, 토큰 리턴형을 바꿀 수 있다. Lexical analyzer 클래스의 이름을 Yylex으로부터 변경할 때는 %class 지시문을 사용한다.

```
%class <name>
```

토큰화 함수의 이름을 yylex으로부터 변경할 때는 %function 지시문을 사용한다.

```
%function <name>
```

토큰화 함수의 리턴형을 Yytoken으로부터 변경할 때는 %type 지시문을 사용한다.

```
%type <name>
```

이들 지시문들을 사용하지 않는 경우, 토큰화 함수명과 리턴형은 기본값으로 Yylex.yylex()와 Yytoken이 사용된다.

Scoping 충돌을 피하기 위해 lexical analyzer의 함수와 변수에 예약된 이름들은 yy로 시작한다.

2.2.10. Default 토큰 형

32-bit primitive 정수형인 int를 토큰화 함수의 리턴형으로 하기 위하여 %integer 지시문을 사용한다.

```
%integer
```

기본값으로, 다음과 코드와 같이 Yytoken이 Yylex.yylex()의 리턴값으로 사용된다.

```
class Yylex { ...
public Yytoken yylex() {
... }
```

%integer 지시문은 위의 코드를 다음 코드와 같이 토큰형을 int로 바꾼다.

```
class Yylex { ...
public int yylex() {
... }
```

이렇게 함으로써 lexical action들이 다음 코드와 같이 integer를 리턴할 수 있게 한다.

```
{ ...
return 7;
... }
```

integer 리턴형을 사용하려면 end of file 코드도 바꿔줘야 한다. 기본값으론, java.lang.Object 클래스나 자식 클래스의 인스턴스들이 Yylex.yylex()로부터 리턴된다. 생성된 Yylex lexer가 수행되는 과정 중에서 end-of-file을 만나게 되면 Yylex.yylex()로부터 null값이 리턴되어야 한다.

Yylex.yylex()의 리턴 값이 int로 바뀌면, null이 더 이상 리턴될 수 없다. 대신, -1의 값을 갖는 Yylex.YYEOF 상수가 리턴된다. %integer 지시문에는 %yyeof가 내포되어 있다

2.2.11. Default 토큰형 II: Wrapped Integer

java.lang.Integer를 토큰화 함수의 리턴형으로 하기 위하여 %intwrap 지시문을 사용한다.

```
%intwrap
```

기본값으로, 다음 코드와 같이 Yytoken이 Yylex.yylex()의 리턴값으로 사용된다.

```
class Yylex { ...
public Yytoken yylex() {
... }
```

%intwrap 지시문은 위의 코드를 다음 코드와 같이 토큰형을 java.lang.Integer로 바꾼다.

```
class Yylex { ...
public java.lang.Integer yylex() {
... }
```

이렇게 함으로써 lexical action들이 다음 코드와 같이 wrapped integer를 리턴할 수 있게 한다.

```
{ ...
return new java.lang.Integer(0);
... }
```

%intwrap 지시문은 다음과 같이 %type 지시문을 사용한 것과 동일한 효과를 갖는다.

```
%type java.lang.Integer
```

Yylex.yylex()의 return type을 java.lang.Integer로 수동으로 바꾸는 것이다.

2.2.12. End-of-File의 YYEOF

%yyeof 지시문은 Yylex.YYEOF 상수를 선언하게 한다. %integer 지시문이 선언된 경우, Yylex.YYEOF가 end-of-file에서 리턴된다.

```
%yyeof
```

이렇게 하면 다음과 같이 Yylex.YYEOF가 선언된다.

```
public final int YYEOF = -1;
```

%integer 지시문은 %yyeof를 내포한다.

2.2.13. 개행문자의 운영체제간 호환성

UNIX 운영체제에서는 개행문자로 '\n' 단일 character를 사용한다. 반면, DOS 기반 운영체제에서는 "\r\n"(carriage return + newline)이 개행문자로 사용된다. %notunix 지시문은 carriage return이나 newline이 개행문자로 인식되도록 한다.

```
%notunix
```

어떤 문자열을 개행문자로 할 것인가의 문제는 자바의 플랫폼 독립성의 중요한 이슈이다.

2.2.14. Character Sets

Default로는 문자 code 중 0에서 127사이를 사용한다. 이 범위를 벗어나는 문자가 들어오면 lexer가 제대로 동작을 하지 못한다.

%full 지시문은 범위를 8 bit으로 나타낼 수 있는 모든 값으로 확장한다.

```
%full
```

이 경우, 생성될 lexical analyzer가 0에서 255사이의 값들을 받아들인다.

%unicode 지시문은 범위를 16 bit으로 나타낼 수 있는 모든 유니코드로 확장한다.

```
%unicode
```

이 경우, 생성될 lexical analyzer가 0에서 2^16-1사이 값들을 받아들인다.

%ignorecase 지시문은 대소문자 구별을 하지 않는 lexer를 생성하게 한다.

```
%ignorecase
```

이 경우, CUP이 모든 문자 집합들을 유니코드의 방법에 따라 확장하여 대소문자를 구분하지 않는다.

2.2.15. 파일 입출력시의 문자 형식

현재 JLex 버전에서 생성되는 lexical analyzer에서는 한 문자 당 한 바이트가 할당되는 아스키 형식의 텍스트 파일만을 지원한다. 하지만, 앞으로의 확장성을 위하여 비록 16 비트의 전체 표현 범위를 모두 지원하지는 않지만, 모든 JLex의 내부 문자 처리에 16 비트 자바 문자형을 사용하고 있다.

2.2.16. Lexical action들에서 발생되는 exception들"

JLex의 3번째 section에 해당되는 정규식 규칙의 action부 코드들은 exception을 throw하거나 다른 함수로부터의 것을 전파시킬 수 있다. 이들 exception을 선언하려면 %yylexthrow{ ... %yylexthrow} 지시문을 사용하면 된다.

```
%yylexthrow{
<exception[1]>[,<exception[1]>,...]
%yylexthrow}
```

이렇게 선언된 exception들은 lexical analyzer 토큰화 함수인 Yylex.yylex() 선언부에 다음과 같은 형태로 복사된다.

```
public Yytoken yylex()
throws <exception[1]>[,<exception[1]>,...]
{
...
}
```

%yylexthrow{ ... %yylexthrow} 지시문안의 자바 코드에서 선언되지 않은 exception을 throw하는 경우 생성된 lexical analyzer 소스 파일이 성공적으로 컴파일 되지 못할 것이다.

2.2.17. End-of-File에서의 리턴값

%eofval{ ... %eofval} 지시문은 end-of-file의 리턴값을 명시하는 역할을 한다. 이 지시문은 lexical analyzer가 end-of-file을 만났을 때 수행되는 Yylex.yylex()안에 넣을 코드를 작성할 수

있게 해준다. 이 코드에서는 토큰화 함수 Yylex.yylex()의 리턴 타입과 일치하는 값을 리턴해야 한다.

```
%eofval{
<code>
%eofval}
```

위와 같은 방법으로 lexical analyzer 클래스가 입력 파일의 끝에 도달하였을 때 Yylex.yylex()의 리턴값을 결정하는 code를 작성할 수 있다. end-of-file에 여러 번 도달하는 경우 그때마다 위의 <code>에서 Yylex.yylex()의 리턴값을 결정한다. 다른 지시문들과 마찬가지로 %eofval{과 %eofval} 지시문은 라인의 처음에 위치해야 한다.

다음은 %eofval{ ... %eofval} 지시문의 사용 예이다. End-of-file의 리턴 값으로 기본값인 null이 아닌 (new token(sym.EOF))를 원하는 경우 다음과 같이 specification 파일에 추가하면 된다.

```
%eofval{
return (new token(sym.EOF));
%eofval}
```

이렇게 하면 Yylex.yylex()가 다음과 같이 생성된다.

```
public Yytoken yylex(){ ...
return (new token(sym.EOF));
... }
```

2.2.18. Implement할 Interface의 명시법

JLex에서는 Yylex 클래스가 특정 interface를 implement하게 할 수 있다. 다음과 같은 선언문을 추가하면 된다.

```
%implements <classname>
```

이렇게 하면 다음과 같은 lexical analyzer 클래스가 생성된다.

```
class Yylex implements classname { ...
```

2.2.19. Lexical analyzer 클래스를 public으로 하는 방법

%public 지시문을 선언하면 JLex가 lexical analyzer 클래스를 public 클래스로 만든다.

```
%public
```

default로는 아무런 클래스 access specifier도 붙지 않아서, 같은 패키지 안에서만 참조 가능하게 한다.

2.3. 정규식 규칙들

JLex specification 파일의 세 번째 부분에는 입력 스트림을 토큰으로 분해하는 규칙들을 열거한다. 이들 규칙들은 정규식과 이에 연결된 action을 설명하는 자바 코드로 구성되어 있다.

하나의 규칙은 Optional 상태 list, 정규식, action의 세 부분으로 나누어지며, 다음과 같은 형식을 갖는다.

```
[<상태s>]<정규식>{<action>}
```

각 부분에 대한 자세한 설명은 이후에 하겠다.

입력 문자열이 하나 이상의 규칙과 일치하는 경우, lexer는 길이가 가장 긴 문자열과 일치하는 규칙을 선택한다. 그래도, 길이가 같은 문자열이 여러 개인 경우 JLex specification에 먼저 명시된 규칙을 선택한다. 결국, specification에서 더 앞에 있는 규칙들이 더 높은 우선권을 갖는다고 볼 수 있다.

JLex specification 파일에 선언된 규칙들은 가능한 모든 입력에 적용될 수 있어야 한다. 만약, lexical anaylzer가 이들 규칙에 적용되지 못하는 입력을 받는 경우 에러가 발생한다.

결국, 모든 입력들은 적어도 하나의 규칙에 적용될 수 있어야 하며, 이것은 다음과 같은 규칙을 제일 마지막에 추가하는 방법으로 보장할 수 있다.

```
. { java.lang.System.out.println("Unmatched input:" + yytext()); }
```

점(.)은 개행 문자를 제외한 모든 입력을 의미한다.

2.3.1. Lexical States

Optional lexical 상태 리스트는 규칙의 앞 부분에 위치하며, 다음과 같은 형식을 따른다.

```
<상태[0][,state[1],stateg[2],...]>
```

상태의 중복 선언은 선택사항이며, 꺽쇄 짝(<>)안에 위치해야 한다. 상태 list는 어떤 초기 상태하에서 규칙이 적용될 수 있는가를 나타낸다.

예를 들어, yylex()가 상태 A일때 호출된 경우, lexer는 상태 list에 A가 포함된 규칙들만을 적용시킬 수 있다.

상태 list가 없는 것은, 모든 상태에서 해당 규칙이 적용 가능함을 나타낸다.

2.3.2. 정규식

White space는 정규식의 종료로 해석되기 때문에 정규식 중에 white space가 포함되면 안된다. 한가지 예외로 개행문자를 제외한 white space는 따옴표로 둘러쌈으로써 자기 자신으로 표현될 수 있다. 예를 들어, " "는 black space로 해석된다.

JLex는 정규식에 0~127사이의 Ascii code를 사용한다.

다음 문자들은 JLex의 정규식에서 특별한 의미를 가지는 metacharacter들이다.

**? * + | ( ) ^ $. [ ] { } " **

다른 문자들은 자기 자신을 나타낸다.

- ef 연속된 정규식은 그들의 연쇄를 나타낸다.
- e|f 수직 바(|)는 양쪽의 정규식을 선택할 수 있음을 나타내므로, e 또는 f를 뜻한다.
- Backslash 뒤에 따라오는 문자열은 다음과 같이 해석된다.
  - \b Backspace
  - \n newline
  - \t Tab
  - \f Formfeed
  - \r Carriage return
  - \ddd 3자리 8진수 숫자에 해당하는 character code
  - \xdd 2자리 16진수 숫자에 해당하는 character code
  - \udddd 4자리 16진수 숫자에 해당하는 Unicode character code
  - \^C Control character
  - \c 이외의 다른 문자가 backslash 뒤에 오는 경우 그 문자 자신을 나타낸다
- $ 달라 기호는 줄의 마지막을 의미한다. 정규식의 마지막에 달라 기호가 있는 경우 그 정규식은 줄의 마지막에서만 적용됨을 의미한다.
- . 개행문자를 제외한 모든 문자. [^\n]과 동일하다.
- "..." 따옴표 안에 metacharacter가 있는 경우, 원래 의미를 잃고 자기 자신을 표현한다. \"는 예외적으로 따옴표 문자 "를 가리킨다.
- {name} 매크로 expansion.
- * Kleen closure로 앞의 정규식의 0번 또는 그 이상의 반복.
- + 앞의 정규식의 한번 이상의 반복. 따라서, e+는 ee*와 동일하다
- ? 앞의 정규식의 0번 또는 한번의 반복.
- (...) 정규식을 묶는다.
- [...] 대괄호는 문자 집합을 나타내며, 집합에 포함된 어느 하나의 문자에 대해 정규식이 적용된다. 대괄호 안의 첫 문자가 ^인 경우 여집합이 되어 대괄호 안에 포함된 것들 이외의 문자에 대해 정규식이 적용된다. 대괄호 안에서는 다음과 같은 다른 metacharacter 규칙이 적용된다.
  - {name} 매크로 expansion
  - "..." 따옴표 안에 metacharacter가 있는 경우, 원래 의미를 잃고 자기 자신을 표현한다. \"는 예외적으로 따옴표 문자 "를 가리킨다.
  - \ backslash뒤의 metacharacter는 원래의 특별한 의미를 잃는다.

- \- 대괄호 안의 첫 글자나 끝 글자가 -인 경우 -는 원래의 특별한 의미를 잃는다.

예) [a-z] 임의의 소문자 [^0-9] 아라비아 숫자를 제외한 모든 문자 [0-9a-fA-F] 임의의 16진수
[\-\\] dash or backslash ["A-Z"] A, dash, or Z [+-] and [-+] '+' or '-'

2.3.3. 연관된 **Action**들

Lexical 규칙에 연관된 Action은 중괄호로 묶여진 자바 코드로 작성한다.

```
{ action }
```

위의 action 부분 자바 코드는 JLex로 생성된 상태-driven lexical analyzer에 복사된다.

action 부분의 문자열이나 주석을 제외한 부분에 있는 열고 닫는 중괄호들의 숫자는 일치해야 한다.

2.3.3.1. **Action**과 재귀호출

Action에서 아무런 값도 리턴되지 않는 경우, lexical analyzer는 입력 스트림으로부터 다음 토큰을 찾고 리턴할 때까지 루프를 돈다.

다음과 같이 하여 yylex를 명시적으로 재귀호출 할 수 있다.

```
{ ...
return yylex();
... }
```

이렇게 하면 다음 토큰을 찾고 해당 값이 리턴될 때까지 재귀적으로 lexical analyzer가 동작한다. 하지만, 주어진 action에서 아무런 값도 리턴하지 않게 함으로써 같은 효과를 얻을 수 있으며, 이렇게 하면 재귀 호출로 인한 overhead를 피할 수 있다.

앞의 코드는 호출하는 함수의 끝에서 재귀호출이 일어나기 때문에 tail recursion이다. 아래 코드는 tail recursion이 아닌 재귀호출의 예이다.

```
{ ...
next = yylex();
... }
```

tail recursion이 아닌 재귀호출의 경우에도 yyline이나 yychar와 같은 변수들의 값이 재귀호출 도중 바뀌는 점만 재외한다면 올바르게 동작한다.

2.3.3.2. **State** 전환

JLex 지시문 section에 lexical 상태들이 선언된 경우, 정규식 action에 상태 전환을 다음과 같은 형태로 지시할 수 있다.

```
yybegin(상태);
```

void형 yybegin()함수는 상태 이름인 상태를 넘겨받아서 해당 상태로 전환을 한다.

yybegin의 parameter는 JLex 지시문 section에서 선언되어야 하며, 그렇지 않은 경우 생성된 소스의 컴파일 과정에서 에러가 발생한다. 예외적으로 YYINITIAL은 암시적으로 JLex에 의해 선언되어 있기 때문에 추가의 선언문이 필요 없다. 생성된 lexer는 YYINITIAL 상태로 부터 시작되며 상태 전환이 있기 전까지 이 상태에 머문다.

2.3.3.3. 사용 가능한 **Lexical** 변수들

아래의 변수들은 Yylex 클래스에 내부적으로 선언되는 것들로 lexical 규칙의 action 부에서 사용 가능하다.

표 **1.**

Variable or Method	Activation Directive	Description
java.lang.String yytext();	Always active.	정규식과 일치된 입력 스트림의 문자열 부분
int yychar;	%char	정규식과 일치된 부분의 입력 스트림에서의 zero-based 문자 index
int yyline;	%line	정규식과 일치된 부분의 입력 스트림에서의 zero-based 행번호

3. 생성되는 **Lexical Anaylzer**

문법에 맞게 작성된 specification으로 JLex를 실행시키면, 해당 기능을 수행하는 변환된 lexical analyzer 자바 소스 코드를 얻을 수 있다.

생성된 lexical analyzer는 Yylex 클래스에 구현된다. 여기엔 토큰화할 입력 스트림을 인자를 받는 두개의 생성자가 존재한다. 입력 스트림으로는 java.io.InputStream이나 java.io.Reader(예. StringReader)를 사용할 수 있다. JDK1.0의 java.io.InputStream은 유니코드를 제대로 읽지 못하기 때문에, 유니코드를 사용하려는 경우 java.io.Reader를 사용해야 함에 유의하자.

lexer는 Yylex.yylex()를 사용하여 접근 가능하다. 이 함수는 입력 스트림에 존재하는 바로 다음 토큰을 리턴한다. 리턴형은 Yytoken이며 다음과 같이 함수 형태가 정의되어 있다.

```
class Yylex { ...
public Yytoken yylex() {
... }
```

사용자는 Yytoken 형을 정의해야만 하며, 이것은 JLex specification의 첫 번째 section에서 쉽게 할 수 있는 작업이다. 예를 들어, Yylex.yylex()가 integer의 wrapper를 리턴하는 경우, 첫 번째 %% 앞에 다음과 같은 코드를 작성할 수 있다.

```
class Yytoken { int field; Yytoken(int f) { field=f; } }
```

이렇게 한 뒤, lexical action에서 다음과 같이 wrapped integer를 리턴할 수 있다.

```
{ ...
return new Yytoken(0);
... }
```

마찬가지로, 사용자 코드 section에서 각 토큰형에 해당하는 상수를 다음과 같이 정의할 수 있다.

```
class TokenCodes { ...
public static final STRING = 0;
public static final INTEGER = 1;
... }
```

이렇게 한 뒤, 이들 중 하나를 다음과 같이 리턴할 수 있다.

```
{ ...
return new Yytoken(STRING);
... }
```

이상은 아주 간단한 예들로, 실제로는 훨씬 많은 정보를 담을 수 있는 토큰 클래스를 만들어야 하는 경우가 많다.

Yylex.yylex()에서 리턴되는 토큰형을 보다 다양하게 하기 위해서 여러가지 객체 지향적인 기법들을 사용하는 것이 좋다. 예를 들어, 상속을 사용하면 하나 이상의 토큰형을 사용할 수 있게 된다. 문자열과 정수들에 대해 토큰형을 구별하고 싶은 경우 다음과 같이 하면 된다.

```
class Yytoken { ... }
class IntegerToken extends Yytoken { ... }
class String extends Yytoken { ... }
```

이렇게 한 뒤, IntegerToken과 StringToken 형을 구별하여 사용할 수 있다.

lexical analyzer 클래스명, 토큰화 함수명과 리턴형은 JLex 지시문을 통하여 변경될 수 있다. 2.2.9 section에 보다 자세한 설명이 있다.

4. 성능

JLex에 의해 생성된 lexical analyzer와 수동으로 작성된 lexical analyzer의 성능을 비교한 벤치마킹 결과가 있다. 실험 시 사용된 예제 입력은 간단한 프로그래밍 언어였으며, 수동으로 작성된 lexical analyzer도 자바로 작성이 되었다.

두개의 간단한 프로그래밍 언어 소스 파일에 대해 각각의 lexical analyzer를 동작시키고, 처리시간을 측정하였다. 각각의 lexical analyzer는 자바로 작성된 driver에 의해 수행되었다.

JLex로 생성된 lexical analyzer가 더 빠르다는 것이 다음 실험 결과와 같이 판명되었다.

표 **2.**

Size of Source File	**JLex-Generated Lexical Anaylzer**	**Hand-Written Lexical Analyzer**
177 lines	0.42 seconds	0.52 seconds
897 lines	0.98 seconds	1.28 seconds

JLex lexical analyzer가 수동으로 작성된 lexer보다 충분히 빠른 속도를 가지고 있음을 알 수 있다.

JLex와 같은 프로그램에 의해 만들어진 table-driven lexical analyzer들의 가장 큰 문제점 중 하나는 수동으로 작성된 것에 비해 속도가 느리다는 점이었다. 그러므로, 이것은 JLex lexical analyzer의 상대적인 속도를 입증하는 중요한 실험 결과로 볼 수 있다.

5. 현재의 구현상의 초점들

5.1. 구현되지 않은 기능들

다음들은 JLex에서 구현되지 않은 기능의 목록이다. (완전한 목록이 아닐 수도 있다.)

- 정규식의 lookahead 연산자가 구현되지 않았으며, 특별한 몇몇 정규식의 metacharacter들을 포함하지 못하였다.
- 행의 시작을 나타내는 연산자(^)의 동작이 비정상적인 것으로 여겨진다. 이 연산자가 사용된 정규식이 적용되었을 때 앞의 개행문자가 버려지는 일이 발생한다.

5.2. 유니코드와 아스키 코드

Ansi C에서 8비트 문자형(char)를 사용하는데 비해, 자바에서는 16비트 문자형과 유니코드를 지원하며, 기본적으로 제공되는 String 클래스에서 유니 코드 문자를 사용하고 있다.

1.2.5 버전에 들어서 JLex는 JDK 1.1의 Reader와 Write 클래스를 lexical analyzer의 입력 파일의 입출력에 사용하기 시작했다. 이것은 유니코드를 입출력 모두에서 사용 가능하다는 뜻이다. 물론, 생성된 scanner가 유니코드를 제대로 사용할 수 있게 하기 위해서는, scanner의 생성자의 인자로 java.io.Reader 인스턴스를 주어야 하며, OS 고유의 형식을 유니코드로 정확히 바

꾸어 줄 수 있는 Reader를 사용해야 한다. %unicode 지시문도 선언해 주어야 한다. (section 2.2.14 참고)

5.3. 상태 목록에서의 쉼표

상태 선언 목록과 lexical 규칙 사이의 쉼표는 선택사항이다. 이들은 쉼표 분리자 없이도 공백 문자를 사용하여 올바르게 해석 가능하다.

5.4. 구현되지 않는 기능 중 앞으로 개선할 계획 중인 것들

다음은 곧 JLex에 구현될 수 있는 것들이지만, scope나 성능에 미치는 악효과 등을 고려하여 아직 구현되지 않은 기능들이다.

- lexical action에서의 열고 닫는 횟수가 맞지 않는 주석 표시자의 검출
- 매크로 정의문에서의 순환

6. Credits and Copyrights

6.1. Credits

The treatment of lexical analyzer generators given in Alan Holub's Compiler Design in C(Prentice-Hall, 1990) provided a starting point for my implementation.

Discussions with Professor Andrew Appel of the Princeton University Computer Science Department provided guidance in the design of JLex

Java is a trademart of Sun Microsystems Incorporated.

6.2. Copyright

JLex COPYRIGHT NOTICE, LICENSE AND DICSCLAIMER.

Copyright 1996 by Elliot Joel Berk.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copyes and that both the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the name of Elliot Joel Berk not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Elliot Joel Berk discaims all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall Elliot Joel Berk be liable for any speicial, indirect or consequntial damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortous action, arising out of or in connection with the use or performance of this software.