컴파일러 구성을 위한 대표적 소프트웨어 도구인 Lex(또는 Flex)와 Yacc(또는 Bison) 입문: Windows 환경에서

김도형(성신여자대학교 IT학부)

0. 소개

(1) Lex와 Yacc은 무엇인가?

컴파일러의 구성을 도와주는 대표적인 소프트웨어 도구들이다. 원래 UNIX의 산실인 벨 연구소(AT&T Bell Laboratories)에서 UNIX 시스템의 유틸리티(utility)로서 개발되었다. 각각의 개발자는 Lex는 Mark Lesk, Yacc은 Steve Johnson이다.

이후 이 도구들의 유용성이 확인되면서 Sun의 Solaris나 Linux와 같은 UNIX 계열의 운영체제는 물론이고, Microsoft Windows 등의 상이한 운영체제에도 이식(porting)되어서 널리 사용되고 있다.

(2) Lex와 Yacc을 어떻게 구하는가?

현재 UNIX 계열 상용(商用) 운영체제의 경우 Lex와 Yacc은 기본 유틸리티의 일부로서 포함되어 있다. 그러나 Lex와 Yacc의 판권(copyright)은 Bell Laboratories(현재 Lucent Technology)가 가지고 있으므로, Linux와 같은 FSF(Free Software Foundation)의 정신을 표방하는 운영체제의 경우는 각각 Lex와 Yacc의 GNU 버전(version)인 Flex와 Bison을 기본 유틸리티로서 갖추고 있다. 그 외도 여러 운영체제를 위한 약간씩 다른 Lex와 Yacc의 버전들이 있다.

Windows 상에서 Cygwin 시스템을 사용한다면 처음 설치할 때 Flex와 Bison을 선택하여 사용하는 것이 가장 UNIX 계열 운영체제와 유사하게 사용할 수 있을 것이다. 이밖에도 Windows 환경에서 Lex와 Yacc 또는 그에 상응하는 유틸리티를 이용하기 위한 몇 가지 다른 방법이 있다. 원래의 Lex나 Yacc과는 달리 결과 파일을 C 언어뿐만 아니라 C++ 언어나 Java 언어로 출력하는 상용 소프트웨어도 시판되고 있다. 또한 Cygwin 시스템과 같이 UNIX 계열 운영체제와 유사한 환경 전체의 설치가 필요 없는 경우, GNU C 컴파일러(gcc)와 Flex, Bison만 Windows 운영체제에 설치하여 사용할 수도 있다.

이 Windows 운영체제용 튜토리얼(tutorial)에서는 가장 간단한 마지막 방법에 기반을 두어 설명하도록 하겠다. (추후 Windows 운영체제를 위한 다른 방법에 대한 내용을 추가할 계획이다.)

(3) Windows 운영체제에 GNU C 컴파일러와 Flex, Bison 설치하기

먼저 GNU C 컴파일러를 설치한다. 이것을 설치하는 편한 방법 중 하나가 유명한 무료 C/C++ 프로그

램용 통합 개발 환경(IDE; integrated development environment)인 Bloodshed 사의 Dev-C++를 설치하는 것이다(관련 사이트 링크: http://www.bloodshed.net/devcpp.html). 원래 Dev-C++가 GNU C 컴파일러를 기본으로 하여 그 위에 사용자 편의를 위한 IDE 인터페이스를 뒤집어씌운 것이기 때문이다. Dev-C++를 설치하면(주의: 이것이 설치되는 폴더(folder)까지 이르는 경로명(path name)에 공백이 없도록 한다. 보통 'C:\Dev-Cpp'에 설치한다) 그 설치 디렉터리(directory) 아래 'bin'이라는 하위 디렉터리가 만들어지고 이 안에 GNU C 컴파일러('gcc.exe')와 C++ 컴파일러('g++.exe')가 설치된다.

그 다음에는 Flex와 Bison을 설치한다. (비록 이렇게 서술하지만 이 세 프로그램(GNU C 컴파일러, Flex, Bison)의 설치 순서는 바뀌어도 상관없다.)

Flex를 다운로드 받는다(관련 사이트 링크: http://gnuwin32.sourceforge.net/packages/flex.htm). 설치는 매우 간단하여 다운로드 받은 설치 파일을 실행시킨 후 몇 번 '동의' 또는 '다음' 버튼을 누르기만 하면 된다(주의: Dev-C++와 마찬가지로 이것이 설치되는 폴더까지 이르는 경로명에 공백이 없도록 한다. 보통 'C:\GnuWin32'에 설치한다).

Bison도 다운로드 받아서(관련 사이트 링크: http://gnuwin32.sourceforge.net/packages/bison.htm) Flex와 같은 요령으로 설치한다(주의: 앞의 두 프로그램과 마찬가지로 이것이 설치되는 폴더까지 이르는 경로명에 공백이 없도록 한다. 경로명에 공백이 있는 경우 실제 Bison을 사용할 때 오류가 발생하는 것을 확인했으니 특히 주의해야 한다. 보통 Flex와 함께 'C:\GnuWin32'에 설치한다).

세 프로그램의 설치를 마치면 시스템의 환경 변수들 중 'PATH' 변수의 값에 GNU C 컴파일러와 Flex, Bison의 실행 파일이 있는 경로를 추가한다. 즉, 예컨대 'C:\Dev-Cpp\bin;C:\GnuWin32\bin'을 추가내지 ('PATH' 변수가 없었던 경우) 새롭게 등록하는 것이다.

명령 프롬프트(command prompt) 창을 연다('모든 프로그램' → '보조 프로그램' → '명령 프롬프트' 메뉴를 선택하여 실행하거나 실행 창에서 'cmd(.exe)'를 입력하여 바로 실행시킨다). 이 창에서 'flex --version'을 입력하여 설치된 Flex의 버전을 확인한다(최신 버전은 2.5.4). 'bison --version'을 입력하여 설치된 Bison의 버전을 확인한다(최신 버전은 2.4.1). 'gcc --version'을 입력하여 설치된 GNU C 컴파일러의 버전을 확인한다(최신 버전은 3.4.2이나 Dev-C++ 안정 버전에 포함된 것은 3.3.1).

설치의 마지막으로 Flex와 Bison이 필요로 하는 라이브러리 파일을 GNU C 컴파일러가 찾을 수 있도록 적절한 디렉터리에 복사한다. Flex와 Bison이 설치된 디렉터리(예컨대 'C:\GnuWin32')에 가면 여러 하위 디렉터리가 있는데, 그 중 'lib' 속에 'libfl.a'와 'liby.a'가 포함되어 있고 이것들이 각각 Flex와 Bison의 라이브러리 아카이브(archive) 파일이다. 이것들을 GNU C 컴파일러의 라이브러리 디렉터리(예컨대 'C:\Dev-Cpp\lib')에 복사한다.

Flex와 Bison을 사용하는 일반적인 형태는 다음

```
C:\Users\JohnDoe> flex test.l
C:\Users\JohnDoe> bison test.y
```

과 같다. (Flex 입력 파일의 이름은 'test.l', Bison 입력 파일의 이름은 'test.y'라고 가정한다.) 이렇게 하면 현재 작업하는 디렉터리에 Flex와 Bison의 실행 결과 파일이 만들어진다. 보다 자세한 설명은 뒤에서 여러 예제를 직접 보면서 할 것이다.

1. Lex(Flex)와 Yacc(Bison)의 작동

(1) Lex(Flex)와 Yacc(Bison)의 역할

Lex(Flex)와 Yacc(Bison) 모두 사용자가 작성한 명세(specification) 파일을 처리하여 각각 기본적으로 C 언어로 작성된 어휘 분석 프로그램(lexical analyzer 또는 scanner)과 구문 분석 프로그램(syntax analyzer 또는 parser)을 만들어낸다(이 결과 프로그램을 C++ 언어 또는 Java로 내는 확장 상용 버전도 있다). 사용자는 이 결과 파일을 컴파일하여 단독으로, 또는 컴파일러나 다른 용도의 더 큰 프로그램의 일부로 사용하게 된다.

그리 중요한 점은 아니나, 오리지널 Lex와 Yacc 프로그램이 결과로 만들어내는 파일의 이름은 각각 'lex.yy.c'와 'y.tab.c'로 고정되어 있었다. 그러나 GNU 버전인 Flex와 Bison에서는 입력 파일의 이름 뒤에 확장자로 각각 'yy.c'와 'tab.c'를 붙여서 결과 파일의 이름을 정할 수 있도록 수정되었다.

각 프로그램의 입력으로 들어가는 명세 파일의 내용은 그 목적에 비추어 보면 명백하게 예상할 수 있다. 어휘 분석기의 경우, 입력으로 기호(대개 문자)들의 스트림(stream)을 받아 그 기호들로 이루어진 스트링(string)들의 스트림을 출력으로 내보내므로(이 기호 스트링을 토큰(token)이라고도 부르고, 이러한 이유로 scanner를 tokenizer라고 부르기도 한다), Lex(FLex) 입력 명세에는 입력 스트림으로부터 한 단위로 모아야 하는 패턴들이 규정되어 있어야 한다. 구문 분석기의 경우는 기호 스트링들의 스트림을 입력으로 받아 그 스트링들이 주어진 문법에 부합되는지의 여부를 판단하고 부합할 때는 출력으로 그 스트링들의 파스 트리(parse tree) 또는 구문 트리(syntax tree)를 내보낸다(출력은 컴퓨터 내부에서 꼭 명시적으로 트리 형태를 가질 필요는 없으며, 구문 분석 과정에서 묵시적으로 구성되어 이용될 수도 있다). 따라서 Yacc(Bison) 입력 명세에는 문법의 생성 규칙(production rule)들이 규정되어 있어야 한다.

(2) Lex(Flex)와 Yacc(Bison)의 입력 파일 형식의 얼개

두 프로그램의 입력 파일 모두 크게 세 부분으로 구성되어 있으며, 각 부분의 역할도 비슷하다. 첫 번째 부분은 선언(declaration)이나 정의(definition)가 포함되며, 두 번째 부분은 본론에 해당하는 것으로서 각 프로그램이 수행하는 일에 대한 규칙(보통 번역 규칙(translation rule)이라고 부른다)을 기술하고, 세 번째 부분은 보조 프로시저(auxiliary procedure) 또는 지원 프로그램(supporting routines)을 담고 있다. 이 세 부분들 중 꼭 있어야 하는 필수 부분은 두 번째 부분이고, 첫 번째와 세 번째 부분은 필요 없는 경우 생략이 가능하다. 각 부분은 연속된 두 개의 '%' 기호로 구분된다.

첫 번째와 세 번째 부분에 대한 설명은 뒤에 예제를 통해 하기로 하고, 가장 핵심이 되는 두 번째 부분에 대해 간략하게 언급하고자 한다.

Lex(Flex)의 경우 이 두 번째 부분은 아래

```
패턴1   { 동작1 }
패턴2   { 동작2 }
  …        …
패턴n   { 동작n }
```

와 같은 형식을 가지고 있는데, 앞부분의 패턴(pattern)이 입력 스트림에서 발견되면 뒷부분에 있는 동작(action)을 실행하라는 의미이다. 동작은 C 언어로 된 문장들로서, 하나의 문장이면 중괄호(curly braces)가 필요 없다. 패턴은 스트링 패턴을 기술하는 데 보편적으로 사용되는 정규 표현(regular expression)을 Lex에서 확장한 것을 사용한다.

Yacc(Bison)의 경우 이 두 번째 부분은 아래

```
생성 규칙1의 좌변 : 의미 동작이 추가된 생성 규칙1의 우변
생성 규칙2의 좌변 : 의미 동작이 추가된 생성 규칙2의 우변
        …                    …
생성 규칙n의 좌변 : 의미 동작이 추가된 생성 규칙n의 우변
```

와 같은 형식을 가지고 있는데, 생성 규칙의 우변에 있는 문법 기호(grammar symbol)들 사이의 필요한 곳에 중괄호에 둘러싸인 의미 동작(semantic action)이 삽입되어 있는 형태이다. 즉, 구문-인도 번역(syntax-directed translation)의 한 종류인 번역 방략(translation scheme) 형태인 것이다. 의미 동작은 Lex(Flex) 입력 파일에서의 동작과 마찬가지로 역시 C 언어로 된 문장들이다.

2. 예제들을 통한 Lex(Flex) 설명

(1) 첫 번째 예제: 파일의 줄 번호 붙이기 — 버전 1

```
%{
/*
 *   line numbering 1
 */

int         lineno = 1;
%}

%%

\n          { lineno++; ECHO; }
^.*$        printf("%d\t%s", lineno, yytext);
```

예제 1. 파일 이름 'ln1.l': 공백 줄은 건너뛰면서 줄 번호 붙이기

먼저, 상기 파일의 경우 Lex 입력 파일 형식에서 세 번째 부분이 없고 첫 번째와 두 번째 부분만 있는 상태이다. 세 번째 부분이 없으므로, 두 번째와 세 번째 부분을 구분하는 두 번째 '%%'은 생략되어 있다.

첫 번째 부분에서 '%{'와 '%}'에 둘러싸인 부분을 볼 수 있는데, 이 부분은 Lex(Flex)에 의해 별다른 처리가 되지 않고 내용 그대로(in verbatim)가 결과 파일인 'lex.yy.c'(이 이름은 Flex의 경우 '-o' 옵션과 함께 사용자가 원하는 대로 지정하여 바꿀 수 있다)에 포함된다(being dumped). 결과적으로 'lineno'란 전역(global) 변수를 어휘 분석기 프로그램에서 사용할 수 있게 된다. 이 변수는 입력 파일의 줄 번호를 세는 역할을 하게 된다.

두 번째 부분에는 두 개의 패턴이 규정되어 있다.

첫째는 '\n'인데, 글자 그대로 줄바꿈(new line) 문자만이 이 패턴에 부합한다. 그리고 그 패턴이 입력에서 인식되면 해야 될 동작으로는 'lineno' 변수를 1 증가시키고 'ECHO'란 이름의 매크로(macro)(이 매크로는 Lex(Flex)에 의해 정의되어 있는데, 역할은 지금 패턴에 부합한다고 인식된 토큰을 출력하는 것이다. 지금 경우는 그 인식된 토큰은 '\n'이다. 따라서 출력에서 줄이 바뀔 것이다)를 호출하는 것이다.

둘째는 '^.*$'인데, 여러 가지 Lex(Flex) 내의 메타-문자(meta-character)가 사용되고 있다. 제일 앞의 '^'은 입력 파일의 한 줄에서 제일 앞을 의미한다(그렇지만 제일 앞에 있는 특정 기호를 의미하는 것은 아니다). 마지막에 있는 '$'은 반대로 한 줄의 제일 끝을 의미한다(역시 제일 끝에 있는 특정 기호를 의미하는 것은 아니다). 이들 중간에 있는 '.*'에서 '.'은 Lex(Flex)에서 줄바꿈 문자를 제외한 임의의 문자를 의미한다. 그리고 '*'은 표준적인 정규 표현에서의 의미 그대로, 그 앞에 있는 패턴에 부합하는 스트링(기호는 길이가 1인 스트링이므로 당연히 포함)이 아예 없든지 아니면 임의 횟수 반복될 수 있음을 의미한다(reflexive transitive closure). 결국 이것들을 종합하면, 둘째 패턴은 입력 파일에서 임의의 한 줄에서 마지막에 있는 줄바꿈 문자를 제외한 나머지 줄 전체와 부합한다. 그리고 그 패턴이 입력에서 인식되면 줄 번호를 담고 있는 'lineno' 변수의 값을 출력하고 이어서 'yytext'를 출력하는데, 'yytext'는

Lex(Flex)에서 토큰을 인식하는 과정에서 그것을 임시로 저장하는 버퍼(buffer)로서 문자 배열(character array)이다. 결과적으로 패턴에 부합된다고 인식된 토큰을 출력하는 셈이다. (첫째 패턴에서 본 'ECHO'와 'printf("%s", yytext)'는 동일한 결과를 만든다. 인식된 토큰을 출력하는 이런 일이 워낙 빈번하게 사용되므로, Lex(Flex)에서 그것을 위한 축약형으로 'ECHO' 매크로를 정의하여 제공하는 것이다.)

이제 상기 Lex(Flex) 입력 파일을 사용해 보도록 하자. 다음

```
C:\Users\JohnDoe> flex ln1.l
C:\Users\JohnDoe> gcc -o ln1 lex.yy.c -lfl
```

과 같은 순서로 실행시킨다(사용한 플랫폼은 32비트 Windows 7이고, 'C:\Users\JohnDoe>'는 명령 프롬프트(command prompt)이다). Flex 프로그램을 파일 'ln1.l'을 입력으로 하여 실행시킨다. 그 결과로 현재 디렉토리(directory)에 'lex.yy.c' 파일이 생긴다. 이 C 프로그램 파일을 컴파일한다. 이때 주의할 점은 Flex 라이브러리를 컴파일할 때 링크시켜야('-lfl') 무사히 컴파일되어 실행 파일이 만들어진다.

이렇게 만들어진 실행 파일에 아래

```
day := (1461*y) div 4 + (153*m+2) div 5 + d;

if a then c := 1;

while (c)
   do c := c - 1
```

와 같은 입력 파일(파일 이름 'data.p'. Pascal 언어의 구문으로 된 간단한 몇 문장을 포함하고 있다)을 다음

```
C:\Users\JohnDoe> ln1 < data.p
```

과 같이 실행시키면 아래

```
1       day := (1461*y) div 4 + (153*m+2) div 5 + d;

3       if a then c := 1;

5       while (c)
6          do c := c - 1
```

와 같은 결과가 나온다. 결과에서 보듯이 '\n' 문자만으로 이루어진 빈 줄에 대해서는 줄 번호가 없이 빈 줄만 출력된다.

(2) 두 번째 예제: 파일의 줄 번호 붙이기 — 버전 2

Lex(Flex) 입력 파일은 다음

```
%{
/*
 *      line numbering 2
 */

int     lineno = 0;
%}

%%

^.*\n   printf("%d\t%s", ++lineno, yytext);
```

예제 2. 파일 이름 'ln2.l': 공백 줄을 포함하여 모든 줄에 줄 번호 붙이기

과 같다. 예제 1과 마찬가지로, Lex(Flex) 입력 파일 형식에서 세 번째 부분은 없는 상태이다. 예제 1에서 설명한 내용만으로도 예제 2를 이해하는 데는 어려움이 없을 것이다.

예제 1에서처럼 Flex를 실행시킨 뒤 만들어진 'lex.yy.c' 파일을 컴파일하여 만든 실행 파일에 'data.p' 파일을 입력으로 줘서 실행시키면 아래

```
1       day := (1461*y) div 4 + (153*m+2) div 5 + d;
2
3       if a then c := 1;
4
5       while (c)
6          do c := c - 1
```

와 같은 결과가 나온다. 보다시피 모든 줄에 줄 번호가 붙여져 있다.

(3) 세 번째 예제: 파일의 줄 수, 단어 수, 글자 수 세기

Lex(Flex) 입력 파일은 다음

```
%{
/*
 *      word count
 */

int nchar, nword, nline;
%}

%%

\n              ++nchar, ++nline;
[^ \t\n]+       ++nword, nchar += yyleng;
.               ++nchar;

%%

int main(void)
{
        yylex();
        printf("%d\t%d\t%d\n", nchar, nword, nline);
        return 0;
}
```

예제 3. 파일 이름 'wc.l': 입력 파일의 줄 수, 단어 수, 글자 수 세기

과 같다. 상기 파일의 경우 Lex(Flex) 입력 파일 형식의 세 부분을 모두 갖추고 있음에 유의하자.

먼저 첫 번째 부분을 보면, 각각 글자 수, 단어 수, 줄 수를 저장하기 위한 세 변수 'nchar', 'nword', 'nline'이 선언되어 있다. 이 선언은 주석문과 함께 '%{', '%}' 사이에 위치하므로 변경 없이 고스란히 Lex의 결과 파일인 'lex.yy.c'에 전역 변수로 포함된다.

두 번째 부분을 보면 세 개의 패턴과 그에 대한 동작이 규정되어 있다.

첫째는 '\n'으로서 입력 파일에 포함되어 있는 줄바꿈 문자만이 부합한다. 따라서 글자 수와 줄 수가 1씩 증가한다.

둘째는 '[^ \t\n]+'인데 몇 가지 Lex의 메타-문자들이 사용되고 있다. 대괄호('['와 ']')는 문자 클래스(character class)를 나타낸다. 예컨대 패턴 '[ab]'는 스트링 "a"나 "b"가 부합하고, 패턴 '[\t\n]'은 스트링 " "(공백 문자로 이루어진 스트링)이나 "\t"이나 "\n"이 부합한다. Lex에서 '^' 문자는 줄의 제일 앞을 나타내기도 하나, 대괄호 안에서 사용되면 '후속 문자들을 제외한 다른 문자들'이라는 의미를 가진다. 따라서 패턴 '[^ \t\n]'은 공백 문자와 탭(tab) 문자 그리고 줄바꿈 문자를 제외한 다른 임의의 문자들이 부합한다. 마지막으로 '+'는 거의 표준적인 정규 표현으로서 그 앞에 있는 패턴에 부합하는 스트링이 한 번 이상 임의 횟수 반복될 수 있음을 의미한다(positive closure). 결국 종합하면 둘째 패턴은 임의 문자들로 이루어진 임의 길이 스트링(중간에 공백 문자나 탭 문자나 줄바꿈 문자로 끊어지지 않은)에 부합

한다. 곧 '단어'라고 볼 수 있다. 그러므로 단어 수를 1 증가시키고 글자 수는 단어의 길이만큼 증가시켜야 한다. 즉 지금 이 패턴에 부합된 실제 스트링의 길이만큼 'nchar' 변수를 증가시켜야 하는 것이다. Lex(Flex)에서는 명시된 패턴에 부합되어 인식한 입력 스트링의 길이를 담고 있는 변수를 미리 제공하고 있는데 바로 'yyleng'이다.

셋째는 '.'인데 앞에서 설명했듯이 줄바꿈 문자를 제외한 임의 문자가 부합하는 패턴이다. 첫째 패턴과 둘째 패턴에서 부합되지 않는 입력은 이제 ' '와 '\t'이다. 따라서 글자 수만 1 증가시켜 주면 된다.

세 번째 부분을 보면 'main' 함수가 정의되어 있다. 우리가 Lex(Flex)로부터 만들어진 'lex.yy.c' 파일을 열어보면 어휘 분석을 담당하는 함수로 'yylex'가 만들어져 있다. C 프로그램 실행에 필수인 'main' 함수 역시 정의되어 있는데, 이 기본(default) 'main' 함수가 하는 일은 단순히 'yylex' 함수를 호출하는 것이 전부다. 이 'yylex' 함수가 실행되면서 입력의 글자 수, 단어 수, 줄 수 등을 두 번째 부분에서 지시된 대로 계산할 터이나, 기본 'main' 함수는 그 결과를 출력조차 하지 않는다. 따라서 이 출력 기능을 추가한 'main' 함수를 사용자가 새롭게 정의해야 한다. 세 번째 부분의 'main' 함수는 그 역할을 하는 것이다. 이렇게 사용자가 정의한 'main' 함수가 제공되는 경우 자동적으로 Lex(Flex)의 기본 'main' 함수는 가려진다.

예제 1과 2에서처럼 Lex(Flex)를 실행시킨 뒤 만들어진 'lex.yy.c' 파일을 컴파일하여 만든 실행 파일에 'data.p' 파일을 입력으로 줘서 실행시키면 아래

```
92      25      6
```

와 같은 결과가 나온다. (실습 플랫폼이 UNIX 계열이라면, 이것이 제대로 된 결과인지를 확인하기 위해 굳이 'data.p' 파일의 글자 수, 단어 수, 줄 수를 셀 필요는 없다. UNIX 시스템의 기본 명령 중에 'wc'가 있는데 이것이 바로 입력의 글자 수, 단어 수, 줄 수를 세는 기능을 가지고 있다. 물론 Linux 시스템에서도 기본적으로 지원된다. 이 명령에 인자로 'data.p' 파일을 주고 실행시키면 결과는 다음

```
$ wc data.p
 6 25 92 data.p
```

과 같다. 보다시피 글자 수, 단어 수, 줄 수의 순서만 바뀌었을 뿐 동일한 결과를 보여주고 있다.)

3. 예제들을 통한 Yacc(Bison) 설명

(1) 첫 번째 예제: 중위(infix) 수식을 후위(postfix) 수식으로 변환 — 버전 1

Yacc(Bison) 입력 파일은 다음

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token DIGIT

%%

line   : expr '\n'    { putchar('\n'); }
       ;

expr   : expr '+' term { putchar('+'); }
       | expr '-' term { putchar('-'); }
       | term
       ;

term   : DIGIT { printf("%d", yylval); }
       ;

%%

int yylex()
{
   int c;
   while (1) {
      c = getchar();
      if (c == ' ' || c == '\t');
      else if (isdigit(c)) {
         yylval = c - '0';
         return DIGIT;
      }
      else return c;
   }
}

int main()
{
   if (yyparse() == 0) printf("파싱 성공!\n\n");
   else printf("파싱 실패!\n\n");
}
```

예제 4. 파일 이름 'in2po-rec1.y': 단자리 숫자로 이루어진 중위 수식을 후위 수식으로 변환

과 같다. 이 파일에는 세 부분이 모두 있다. 하나씩 살펴보자.

첫 번째 부분에서 '%{'와 '%}'에 둘러싸인 부분은 Lex(Flex)에서와 마찬가지로 Yacc(Bison)이 변경하지 않고 그대로 결과 파일인 'y.tab.c'(오리지널 Yacc의 경우)에 포함시킨다. 결과적으로 Yacc(Bison)이 만든 파서 프로그램에서 C 언어의 표준 헤더 파일인 '<ctype.h>'를 포함시키는 효과를 내고 그 결과 다양한 문자 판별 술어함수(predicate)를 사용할 수 있게 된다. 첫 번째 부분의 나머지에는 Yacc(Bison)의 키워드(keyword) '%token'을 사용한 선언이 하나 있는데, 두 번째 부분에서 사용되는 'DIGIT'이란 기호가 단말(terminal) 기호란 표시이다. 기본적으로 Yacc(Bison)은 두 번째 부분에서 사용되는 모든 식별자(identifier)를 비단말(nonterminal) 기호라고 간주한다. 단말 기호로 Yacc(Bison)이 인식하게 하려면, 홑따옴표(single quote)에 둘러싸인 문자이거나 Yacc 입력 파일 첫 번째 부분에서 '%token'을 사용해 단말 기호라고 선언해야 한다.

두 번째 부분에는 단자리 숫자들이 '+'와 '-'로 엮어진 중위 수식을 표현하기 위한 문법이 기술되어 있고, 거기에 그 중위 수식을 후위 수식으로 변환하기 위한 의미 동작이 추가되어 있다. 곧 번역 방략이 나와 있다는 뜻이다. 이 번역 방략의 내용은 자명하기에(교재에서 학습하기도 한 내용이라) 설명은 생략하고, 문법의 마지막 생성 규칙에 추가된 의미 동작에서 사용되고 있는 변수 'yylval'에 대해서만 언급한다. 이 변수는 Yacc(Bison)에서 미리 정의되어 있는데, 어휘 분석기에서 인식한 토큰을 Yacc(Bison)에 의해 만들어진 파서 쪽으로 반환할 때 추가로 반환할 속성(attribute) 값이 있는 경우 그것을 저장하여 전달하는 데 사용된다. 주로 Yacc(Bison)과 Lex(Flex)를 연동하여 사용할 때 Lex(Flex)에서 만든 어휘 분석기가 토큰을 인식하면서 원하는 값을 저장한 뒤 Yacc(Bison)에서 만들어진 파서 쪽으로 복귀하는 식으로 이용되는 게 일반적이다.

세 번째 부분에는 'main' 함수와 어휘 분석기 함수 'yylex'가 포함되어 있다. Yacc(Bison)에서 만들어진 파서 함수(이름은 'yyparse'로 고정되어 있다)는 파싱 과정에서 필요할 때마다 어휘 분석기에게 토큰을 요구하는데, 어휘 분석기 함수의 이름이 'yylex'라고 무조건 생각한다. 따라서 Lex(Flex)를 이용하여 어휘 분석기를 만드는 경우에는 만들어진 어휘 분석기 함수 이름이 저절로 'yylex'로 고정되어 있으므로 신경 쓸 필요 없으나, 이 예제처럼 수동으로 어휘 분석기를 만드는 경우에도 어휘 분석기 함수 이름은 무조건 'yylex'로 붙여야 한다. 여기서 수동으로 작성된 어휘 분석기 함수 'yylex'의 내용을 보면, 공백 문자와 탭 문자는 그냥 지나가고 숫자는 단자리로 끊어서 'DIGIT'이란 토큰으로 인식한다. 이 'DIGIT' 토큰의 토큰 값은 자동으로 Yacc(Bison)에 의해 정의되므로 사용자는 그냥 'DIGIT'을 반환하면 Yacc(Bison)에서 만들어진 구문 분석기가 알아본다. 단자리 숫자로 이루어진 'DIGIT' 토큰의 경우, 그 단자리 숫자의 실제 정수 값을 구해서 토큰의 추가 속성 값을 저장하는 변수 'yylval'에 넣은 뒤 아울러 파서 쪽으로 복귀한다. 그 외 '+'나 '-'나 '\n'과 같은 한 글자 토큰은 각 문자의 코드(예컨대 ASCII) 값을 토큰 값으로 반환하게 된다. 'main' 함수는 파서 함수인 'yyparse'를 호출하는 것이 주된 임무이다. 호출된 'yyparse' 함수는 필요할 때마다 'yylex' 함수를 불러 토큰을 받아 파싱을 진행한다. 입력에 대해 성공적으로 파싱이 완료되면 'yyparse' 함수는 0을 반환한다.

이제 상기 Yacc(Bison) 입력 파일을 사용해 보자. 다음

```
C:\Users\JohnDoe> bison -y in2po-rec1.y
C:\Users\JohnDoe> gcc -o in2po-rec1 y.tab.c -ly
```

과 같은 순서로 실행시킨다. Bison 프로그램을 파일 'in2po-rec1.y'를 입력으로 하여 실행시킨다. 옵션 '-y'는 Bison을 오리지널 Yacc과 같은 모드(mode)로 실행하라는 뜻이다. 그 결과로 현재 디렉토리에 'y.tab.c' 파일이 생긴다. 이 C 프로그램 파일을 컴파일한다. 이때 주의할 점은 Yacc(Bison) 라이브러리를 컴파일할 때 링크시켜야('-ly') 무사히 컴파일되어 실행 파일이 만들어진다.

이렇게 만들어진 실행 파일을 다음

```
C:\Users\JohnDoe> in2po-rec1
9-5+2
95-2+
파싱 성공!
```

과 같이 사용한다. 보다시피 단자리 숫자를 피연산자로 하는 중위 수식을 동일한 의미의 후위 수식으로 변환한다(출력할 때 피연산자들 사이에 공백을 주지 않아서 붙어 나오는 바람에 보기는 다소 불편하다).

(2) 두 번째 예제: 중위 수식을 후위 수식으로 변환 — 버전 2

첫 번째 예제에서 한 것과 동일한 일을 Lex(Flex)와 Bison을 함께 사용하여 하고자 한다. 먼저 Bison 입력 파일은 다음

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token DIGIT

%%

line   : expr '\n'     { putchar('\n'); }
       ;

expr   : expr '+' term { putchar('+'); }
       | expr '-' term { putchar('-'); }
       | term
       ;

term   : DIGIT { printf("%d", yylval); }
       ;

%%

int main()
{
   if (yyparse() == 0) printf("파싱 성공!\n\n");
   else printf("파싱 실패!\n\n");
}
```

예제 5. 파일 이름 'in2po-rec2.y': 단자리 숫자로 이루어진 중위 수식을 후위 수식으로 변환

과 같다. 보다시피 동일한 일을 하는 첫 번째 예제에서의 Yacc(Bison) 입력 파일과 거의 동일하다. 다만 이제 어휘 분석기는 Lex(Flex)에 의해 만들어질 것이므로, 직접 작성했던 어휘 분석기 함수 'yylex' 부분이 삭제되었다.

Lex(Flex) 입력 파일은 다음

```
%{
#include "in2po-rec2.tab.h"
%}

%%

[ \t]+          ;
[0-9]           { yylval = yytext[0] - '0'; return DIGIT; }
[+\-\n]         return yytext[0];
```

예제 6. 파일 이름 'in2po-rec2.l': 단자리 숫자로 이루어진 중위 수식을 후위 수식으로 변환

과 같다. 세 부분들 중 세 번째 부분은 없는 구조이다.

첫 번째 부분을 보면 Lex(Flex)의 결과 파일인 'lex.yy.c'에 그대로 포함되는 부분이 있는데 'in2po-rec2.tab.h'라는 이름의 헤더 파일을 포함시키는 문장이다. 이 파일은 Bison에 의해 만들어지는데(뒤에서 추가 설명한다), Bison에 의해 정의된 토큰의 값이나 타입을 포함하고 있다. 예를 들면 Bison 입력 파일에서 비단말 기호가 아니라 단말 기호, 즉 토큰으로 선언된 'DIGIT'의 토큰 값은 Bison이 자동으로 배당하는데, 그 정의가 이 파일에 포함되어 있는 것이다. 이 파일에서 그 정의가 나오는 근처 부분을 보면 아래

```
…
/* Tokens.  */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
   /* Put the tokens into the symbol table, so that GDB and other debuggers
      know about them.  */
   enum yytokentype {
     DIGIT = 258
   };
#endif
…
```

와 같다. 보다시피 258의 값을 가지는 기호 상수로 정의되어 있다. 이 헤더 파일을 Lex(Flex)의 결과 파일인 'lex.yy.c'가 포함시켜야 Lex(Flex) 입력 파일의 두 번째 부분에 있는 동작에서 'DIGIT'을 토큰 값으로 사용할 수 있는 것이다.

참고로 Bison이 아닌 Yacc을 동일 입력 파일에 대해 실행시켜 만들어지는 'y.tab.h' 파일을 보면 다음

```
#ifndef _yacc_defines_h_
#define _yacc_defines_h_

#define DIGIT 257
#define YYERRCODE 256

#endif
```

과 같다. Bison에 비하면 상당히 간략하지만, 역시 토큰 'DIGIT'은 257의 값을 가지는 기호 상수로 정의되어 있다.

두 번째 부분은 대부분 이미 설명된 내용만으로 이해에 어려움이 없겠지만, 셋째 패턴에 새롭게 나타난 Lex(Flex)의 메타-문자가 있다. '\'가 그것인데, 이것은 그 바로 뒤에 나오는 문자가 Lex(Flex)의 메타-문자이면 그 의미를 상실하고 문자 자체를 나타내도록 하는 효과가 있다. 즉, 일반적인 UNIX-C 커뮤니티의 관례인 이스케이프(escape) 문자로서의 백슬래쉬(backslash) 의미 그대로 사용된다. 그러면 왜 '-' 앞에 백슬래쉬를 썼는가 하는 점에 설명이 필요한데, '-'가 문자 클래스를 나타내는 대괄호 사이에 있으면 문자들의 범위(range)를 뜻하는 메타-문자로 기능하기 때문이다. 예컨대 '[abcde]'와 '[a-e]'는 동일한 패턴을 나타내는 것이다.

이제 이 두 Lex(Flex) 입력 파일과 Bison 입력 파일을 사용해 보도록 하자. 다음

```
C:\Users\JohnDoe> flex in2po-rec2.l
C:\Users\JohnDoe> bison -d in2po-rec2.y
C:\Users\JohnDoe> gcc -o in2po-rec2 lex.yy.c in2po-rec2.tab.c -lfl -ly
```

과 같은 순서로 명령을 실행하여 실행 파일을 만든다. Lex(Flex)를 실행함으로써 어휘 분석기 'yylex'를 포함하고 있는 'lex.yy.c' 파일이 만들어진다. Yacc과 Bison은 거의 비슷하나, 만들어지는 파일 이름이 차이가 난다. Yacc은 실행 결과 'y.tab.c'를 만드나, Bison은 실행 결과 파일 이름이 'Bison 입력 파일의 확장자 앞부분'에 확장자로 'tab.c'를 붙여서 만든 것이 된다. 예를 들어 Bison 입력 파일 이름이 이 예제에서처럼 'in2po-rec2.y'라고 하면 결과 파일 이름은 'in2po-rec2.tab.c'가 되는 식이다. 또 앞서 말한 Yacc(Bison)의 정의를 포함하고 있는 헤더 파일은 기본적으로 만들어지는 것이 아니라, Yacc(Bison)을 실행시킬 때 '-d' 옵션(option)을 줘야 한다. 이 옵션을 가지고 실행되면 'y.tab.c'(Yacc의 경우)나 'in2po-rec2.tab.c'(Bison의 경우) 뿐만 아니라, 'y.tab.h'(Yacc의 경우)나 'in2po-rec2.tab.h'(Bison의 경우)도 함께 만들어낸다. 이 헤더 파일은 우리 Lex(Flex) 입력 파일에서 규정된 것처럼 'lex.yy.c'에 포함될 것이다.

이렇게 만들어진 실행 파일을 다음

```
C:\Users\JohnDoe> in2po-rec2
9-5+2
95-2+
파싱 성공!
```

과 같이 사용한다. 보다시피 단자리 숫자를 피연산자로 하는 중위 수식을 동일한 의미의 후위 수식으로 변환한다.

(3) 세 번째 예제: 간단한 단자리 수 계산기

단자리 숫자들을 피연산자로 하고 더하기와 빼기만 연산자로 허용되는 중위 수식을 받아들여 그 식의 결과 값을 계산하는 계산기 프로그램을 Yacc(Bison)을 이용해 만들고자 한다. Yacc(Bison) 입력 파일은 다음

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token DIGIT

%%

line   : expr '\n'      { printf("%d\n", $1); }
       ;

expr   : expr '+' term  { $$ = $1 + $3; }
       | expr '-' term  { $$ = $1 - $3; }
       | term
       ;

term   : DIGIT          { $$ = $1; }
       ;

%%

int yylex()
{
   int c;
   while (1) {
      c = getchar();
      if (c == ' ' || c == '\t');
      else if (isdigit(c)) {
         yylval = c - '0';
         return DIGIT;
      }
      else return c;
   }
}

int main()
{
   if (yyparse() == 0) printf("파싱 성공!\n\n");
   else printf("파싱 실패!\n\n");
}
```

예제 7. 파일 이름 'calc.y': 단자리 숫자로 이루어진 중위 수식의 값을 계산

과 같다. 일단 큰 뼈대는 예제 4 표에 있는 Yacc(Bison) 입력 파일과 동일하다. 단지 두 번째 부분에 포함된 의미 동작들이 다를 뿐이다.

여기에서 '$$' 또는 '$1'이나 '$2' 등과 같은 표현이 사용되고 있는데 이것은 생성 규칙에 나타나는 문법 기호에 첨부된 속성 값을 나타낸다. '$$'는 생성 규칙의 좌변에 있는 비단말 기호에 부착된 속성 값, '$*i*' (*i* = 1, 2, ...)는 생성 규칙의 우변에 있는 *i*-번째 문법 기호에 부착된 속성 값을 나타내는 방식이다. 이 표현을 이용하면 파싱이 진행되면서 문법 기호들의 속성 값을 사용한 다양한 응용이 가능하다.

Yacc(Bison)은 상향식(bottom-up) 파싱 알고리즘인 LALR 기법을 사용한다. 그러므로 제일 처음 어휘 분석 단계에서 인식된 토큰에 대해 'yylval' 변수를 통해 초기 속성 값을 부여한다. 파스 트리의 잎 노드(leaf node)에 부착된 속성 값은 이 $-표현을 이용해서 파스 트리의 중간 노드(interior node)나 루트 노드(root node)까지 전달될 수 있다. 이것만 이해하면 상기 Yacc(Bison) 입력 파일을 이해하는 데는 문제가 없을 것이다.

이제 상기 Yacc(Bison) 입력 파일을 사용해 보자. 다음

```
C:\Users\JohnDoe> bison -y calc.y
C:\Users\JohnDoe> gcc -o calc y.tab.c -ly
```

과 같은 순서로 실행시켜 실행 파일을 만든다. 이렇게 만들어진 실행 파일을 다음

```
C:\Users\JohnDoe> ./calc
9-5+2
6
파싱 성공!
```

과 같이 사용한다.

(4) 네 번째 예제: 가상 스택 기계를 위한 어셈블리(assembly) 코드로 컴파일하기

컴파일러 과목의 표준적 교재중 하나인 "Compilers: Principles, Techniques, and Tools (by Aho, Sethi, and Ullman)"의 2.8절에는 가상 스택 기계(Abstract Stack Machine)라고 부르는 페이퍼 머신(paper machine)이 간략하게 정의되어 있다.

이제 Lex(Flex)와 Yacc(Bison)을 이용해 Pascal 언어의 간단한 문장들을 이 가상 스택 기계의 어셈블리 코드로 번역하는 일을 해 보자. 이 문장들을 포함하고 있는 파일(좀 과장하자면 Pascal 프로그램 파일)이 앞에서 몇 번 사용된 'data.p'이다. 아래에 다시 보였다.

```
day := (1461*y) div 4 + (153*m+2) div 5 + d;

if a then c := 1;

while (c)
   do c := c - 1
```

Lex(Flex) 입력 파일은 다음

```
%{
/*
 *      추상 스택 기계의 목적 코드 생성 예제
 */

#include <string.h>
#include "y.tab.h"
%}

delim           [ \t\n]
ws              {delim}+
letter          [A-Za-z]
digit           [0-9]
id              {letter}({letter}|{digit})*
number          {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

%%

{ws}            ;
":="            return(ASSIGN);
div             return(DIV);
mod             return(MOD);
if              return(IF);
then            return(THEN);
while           return(WHILE);
do              return(DO);
{id}            { strcpy(yylval.lexeme, yytext); return(ID); }
{number}        { strcpy(yylval.lexeme, yytext); return(NUM); }
.               return(yytext[0]);
```

예제 8. 파일 이름 'stack-m.l': 추상 스택 기계를 위한 Pascal 컴파일러 Lex(Flex) 입력

과 같다.

상기 Lex(Flex) 입력 파일에 대해서 몇 가지 설명이 필요할 듯하다. 첫 번째 부분을 보면 여러 가지 이름('delim', 'ws' 등)을 정의하는 것이 있는데, 보통 정규 정의(regular definition)이라고 부른다. 이것은 두 번째 부분에서 패턴을 기술할 때 사용함으로써 패턴 기술을 알아보기 쉽게 만들기 위한 부분식(subexpression) 정의라고 볼 수 있다. 또 새롭게 나타난 Lex(Flex) 메타-문자가 하나 있는데, '?'이다. 이 문자도 거의 표준적 정규 표현에 가까운데, '?' 앞의 패턴에 해당하는 스트링이 있을 수도 있고 없을 수도 있다(being optional)는 것을 나타낸다. 상기 파일에서는 상수(number) 패턴을 기술하는데 분수 부분이나 지수 부분이 있을 수도 있고 없을 수도 있다는 점을 나타낸 것이다.

토큰의 추가 속성 값을 전달하기 위한 변수 'yylval'의 기본 타입은 'int'이다. 만약 다른 타입의 값을 전달하고 싶다면 Yacc의 입력 파일에서 '%union' 키워드를 이용한다. 뒤의 이 예제를 위한 Yacc(Bison) 입력 파일에서 보게 되겠지만, 현재 예제에서 토큰의 추가 속성 값 타입은 10개의 원소를

가진 문자 배열로 선언되어 있고 그 값에 접근하기 위한 공용체(union) 필드(field) 이름은 'lexeme'으로 정의되어 있다. 따라서 상기 Lex(Flex) 입력 파일을 보면 식별자나 상수와 같은 토큰이 입력에서 인식되면, 그것이 입력에서 나타난 실제 스트링(lexeme)을 'yylval.lexeme'에 복사되어 파서 쪽으로 전달하도록 되어 있다.

이 예제를 위한 Yacc(Bison) 입력 파일은 다음

```
%{
/*
 *      추상 스택 기계의 목적 코드 생성 예제
 */

#include <stdio.h>
#include <string.h>

char *tmp_lbl1, *tmp_lbl2;
%}

%union {
   char lexeme[10];
}

%start list

%token ID NUM DIV MOD ASSIGN IF THEN WHILE DO

%%

list    :       list ';' stmt
        |       stmt
        ;

stmt    :       ID ASSIGN
                        { printf("\tlvalue\t%s\n", $1); }
                expr
                        { printf("\t:=\n"); }
        |       IF expr
                        { tmp_lbl1 = new_lbl_no();
                          printf("\tgofalse\t%s\n", tmp_lbl1); }
                THEN stmt
                        { printf("label\t%s\n", tmp_lbl1); }
        |       WHILE
                        { tmp_lbl1 = new_lbl_no();
                          printf("label\t%s\n", tmp_lbl1); }
                expr
                        { tmp_lbl2 = new_lbl_no();
                          printf("\tgofalse\t%s\n", tmp_lbl2); }
                DO stmt
                        { printf("\tgoto\t%s\n", tmp_lbl1);
```

```
                        printf("label\t%s\n", tmp_lbl2); }
       ;

expr   :       expr '+' term { printf("\t+\n"); }
       |       expr '-' term { printf("\t-\n"); }
       |       term
       ;

term   :       term '*' factor { printf("\t*\n"); }
       |       term '/' factor { printf("\t/\n"); }
       |       term DIV factor { printf("\tdiv\n"); }
       |       term MOD factor { printf("\tmod\n"); }
       |       factor
       ;

factor :       '(' expr ')'
       |       ID { printf("\trvalue\t%s\n", $1); }
       |       NUM { printf("\tpush\t%s\n", $1); }
       ;

%%

char* new_lbl_no(void)
{
   static int lbl_no = 0;
   char buf[4];
   int i, quot;
   char *lbl_header;

   lbl_header = (char *)malloc(5);
   strcpy(lbl_header, "lbl_");
   buf[3] = '\0';
   quot = lbl_no++;
   for (i = 2 - (quot / 10); i >= 0; i--) {
      buf[i] = '0' + quot % 10;
      quot = quot / 10;
   }
   return((char *)strcat(lbl_header, buf));
}

int main(void)
{
   printf("\nCompilation for Abstract Stack Machine Started...\n\n");
   printf("\nAssembly code for Abstract Stack Machine follows...\n\n");
   if (yyparse() == 0)
      printf("\n\nCompilation for Abstract Stack Machine Completed!\n");
   else
      printf("\n\nCompilation for Abstract Stack Machine Failed!\n");
}
```

예제 9. 파일 이름 'stack-m.y': 추상 스택 기계를 위한 Pascal 컴파일러 Yacc(Bison) 입력

과 같다. 상기 파일의 첫 번째 부분에 나오는 '%start' 키워드는 그 다음에 오는 식별자가 두 번째 부분에 기술되어 있는 문법의 시작 기호(start symbol)라는 점을 선언하는 것이다. 만약 이게 없으면 문법의 첫 번째 생성 규칙의 좌변에 있는 비단말 기호를 시작 기호로 Yacc(Bison)은 생각한다. 세 번째 부분에 정의되어 있는 'new_lbl_no' 함수는 과거 만든 레이블(label)과는 다른 새로운 레이블을 만들어서 돌려주는 함수이다.

상기 Lex(Flex) 입력 파일과 Yacc(Bison) 입력 파일을 이용하여 실행 파일은 다음

```
C:\Users\JohnDoe> flex stack-m.l
C:\Users\JohnDoe> bison -yd stack-m.y
C:\Users\JohnDoe> gcc -o stack-m lex.yy.c y.tab.c -lfl -ly
```

과 같이 만든다. 이렇게 만들어진 실행 파일에 입력으로 앞에 나온 'data.p' 파일을 주고 실행하여 그 결과를 'data.asm'이라는 파일('asm'이라는 확장자는 'assembly code'를 나타내는 뜻에서 붙였다)에 다음

```
C:\Users\JohnDoe> stack-m < data.p > data.asm
```

과 같이 저장한다. 이 결과('data.p' 파일에 들어있는 Pascal 프로그램을 추상 스택 기계의 어셈블리 코드로 번역한 결과) 파일 'data.asm'의 내용은 다음

```
Compilation for Abstract Stack Machine Started...


Assembly code for Abstract Stack Machine follows...

       lvalue  day
       push    1461
       rvalue  y
       *
       push    4
       div
       push    153
       rvalue  m
       *
       push    2
       +
       push    5
       div
       +
       rvalue  d
       +
       :=
       rvalue  a
       gofalse lbl_000
       lvalue  c
       push    1
       :=
label  lbl_000
label  lbl_001
       rvalue  c
       gofalse lbl_002
       lvalue  c
       rvalue  c
       push    1
       -
       :=
       goto    lbl_001
label  lbl_002


Compilation for Abstract Stack Machine Completed!
```

과 같다.

4. 맺는 말

이상으로 몇 가지 예제를 이용하여 컴파일러 구성을 보조하는 대표적 소프트웨어 도구인 Lex(Flex)와 Yacc(Bison)의 작동을 살펴보았다.

넓게 보면 모든 프로그램은 정해진 형식의 입력을 처리하여 역시 정해진 형식의 출력을 내보낸다는 측면에서 일종의 컴파일러이다. 이 말은 곧 컴파일러 구성에 사용되는 여러 기법이 다른 분야의 프로그램 작성에도 폭넓게 적용될 수 있음을 의미한다. 마찬가지로 Lex(Flex)와 Yacc(Bison)의 응용 범위도 매우 광범위하게 미칠 수 있다.