내공 있는 프로그래머로 길러주는

컴파일러의 이해

Chapter 13
렉스와 야크

목차

학습목표

- 렉스와 야크의 개념에 대해 이해할 수 있다.
- 렉스의 입력 파일 형식과 야크의 입력 파일 형식에 대해 이해할 수 있다.
- 플렉스와 바이슨의 설치 방법에 대해 이해 할 수 있다.
- 플렉스 사용법에 대해 이해할 수 있다.
- 바이슨 사용법에 대해 이해할 수 있다.

13.1 렉스와 야크

- 프로그래밍 언어와 컴퓨터 구조가 다양해짐에 따라 n개의 언어를 m개의 기계에 구현하려면 n × m개의 컴파일러가 필요
  - 그런데 하나의 컴파일러를 만들 때에도 많은 시간과 노력이 들기 때문에 컴파일러를 만드는 데 도움을 주는 시스템이 필요.
  - 이러한 시스템을 컴파일러 자동화 도구라 하고 컴파일러-컴파일러(compiler-compiler) 또는 컴파일러 생성기(compiler generator)라 일컫는다.
- 어떤 입력을 주면 출력으로 컴파일러를 자동으로 생성해주는 컴파일러-컴파일러 :
  - PQCC(Production-Quality Compiler Compiler) 팀과 ACK(Amsterdam Compiler Kit) 팀에 의해 만들려고 시도했으나 만족할 만한 결과를 얻지는 못 함.
- 반면에 어휘 분석, 구문 분석, 코드 생성 등 컴파일러의 한 단계(phase)를 자동으로 생성해주는 번역기 제작 시스템(translator writing system)은 실제로 제작되었다.
  - 이 가운데 가장 성공적인 도구는 유닉스(UNIX) 시스템에서 사용되는 렉스(Lex)와 야크(YACC, Yet Another Compiler Compiler)이다.

13.1 렉스와 야크

■ 렉스 :

- 렉스는 어휘 분석기를 자동으로 생성해주는 어휘 분석기 생성기(scanner generator)
  - 1975년 벨 연구소의 레스크(Richard Mike Lesk)와 슈미트(Eric Emerson Schmidt)가 개발
  - 렉스는 사용자가 정의한 토큰에 대한 표현인 정규 표현과 수행 코드를 입력 받아 일반 범용 언어인 C로 작성된 어휘 분석기를 출력
  - 어휘 분석기는 입력 프로그램에서 정규 표현에 해당하는 토큰을 찾았을 때, 거기에 결합된 수행 코드를 수행하여 토큰을 구분해내는 일을 한다.

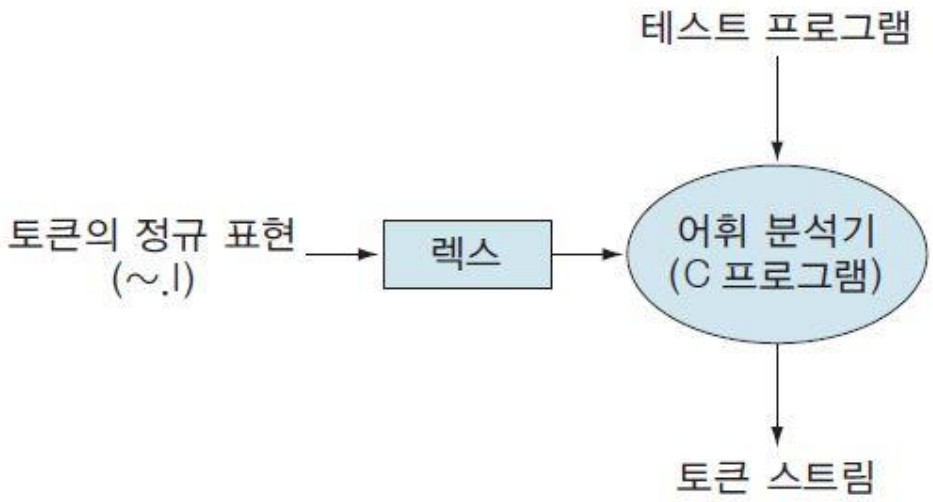

그림 13-1 렉스의 역할

13.1 렉스와 야크

- 현재 사용하고 있는 C 언어에서 식별자를 구분하기 위한 어휘 분석기를 만들고 싶다
  - 가장 먼저 해야 할 일은 렉스의 입력을 만드는 것 - 입력 형태는 2절에서 자세히 설명
  - 식별자에 대해서 첫 자는 영문 소문자·대문 자 또는 밑줄 문자로 시작하고, 둘째 자부터 영문 소문자·대문자, 숫자 및 밑줄 문자가 온다. 식별자의 길이에 제한이 없다고 하면 다음과 같이 간단하게 렉스의 입력을 작성할 수 있다.
  - Letter → [a-zA-Z_]
  - Digit → [0-9]
  - Ident → {Letter}({Letter} | {Digit})*
  - 이 문법을 렉스의 서술 형태에 맞게 작성하고 확장자는 ~.l로 저장
  - 그 다음 렉스를 불러서 실행하면 출력이 생성되는데 lex.yy.c가 어휘 분석기
  - 이 어휘 분석기는 C 언어로 작성된 프로그램이므로, 실행하기 위해 컴파일을 하고 실행하면서 테스트 프로그램을 주면 어휘 분석기가 테스트 프로그램에서 식별자를 구분해낼 것이다.

13.1 렉스와 야크

■ 야크 :

- 야크는 구문 분석기를 자동으로 생성해주는 구문 분석기 생성기(parser generator)
  - 1975년 벨 연구소의 존슨(Steve Johnson)이 주축이 되어 개발한 LALR(1) 구문 분석기 생성기로, 문법 규칙에 대한 수행 코드를 C 언어로 기술하도록 만들어졌다.
  - [그림 13-2]에 야크의 역할 을 나타냈다.

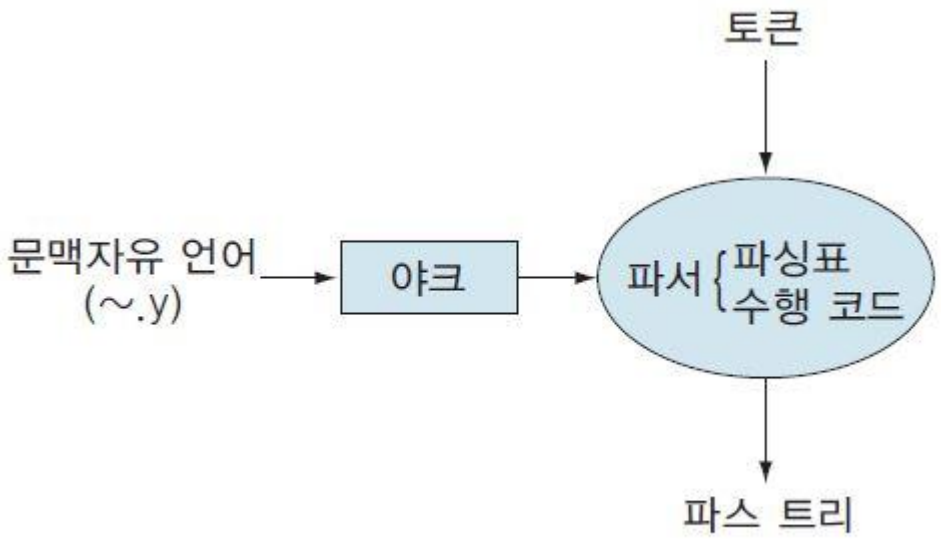

그림 13-2 야크의 역할

13.1 렉스와 야크

- 야크를 사용하려면 야크에 대한 입력을 작성해야 한다.
  - 문법을 야크의 형태에 맞게 서술하고, 확장자는 ~.y로 저장
  - 그런 다음 야크를 불러 실행하면 출력이 생성된다. 출력 중에 y.tab.c가 구문 분석기인데, 구문 분석기는 파싱표와 수행 코드로 구성되고 C 언어로 작성 되어 있다.
  - 따라서 컴파일하고 테스트 프로그램을 입력으로 주면 문법에 맞는 문장인지 아닌 지를 알려줄 것이다.
  - 이 구문 분석기는 LR 파서이다.

13.2 렉스와 야크의 입력 파일 형식

▪ 렉스의 입력 파일 형식

- 렉스는 [그림 13-1]과 같이 렉스의 입력을 먼저 작성하며 입력 프로그램은 파일 확장자가 ~.l이어야 한다.
- 렉스의 입력은 다음과 같이 정의 부분(definition part), 변환 규칙 부분(translation rules part), 사용자 부프로그램 부분(user subprogram part)으로 구성된다.

```
정의 부분
%%
변환 규칙 부분
%%
사용자 부프로그램 부분
```

- 각 부분은 %%로 구분된다. 정의 부분과 사용자 부프로그램 부분은 생략할 수 있지만 변환 규칙 부분은 생략이 불가능하다.

13.2 렉스와 야크의 입력 파일 형식

- 정의 부분은 이름과 일련의 표현식으로 구성되는데, 이름이 식별자이고 표현식은 이름에 해당하는 정규 표현이다.
- 변환 규칙 부분은 표현식과 일련의 수행 코드로 구성된다. 이때 표현식은 정규 표현으로 나타내고, 수행 코드는 표현식을 나타낸 정규 표현과 매칭되었을 때 수행될 일련의 코드로서 C 언어로 작성된다.
- 사용자 부프로그램 부분에서는 토큰에 대한 기호표 처리 루틴(symbol table handling routine) 등과 같이 변환 규칙 부분에서 사용되는 부프로그램이 사용자에 의해 작성된다.
- 정의 부분은 자료 구조나 변수 및 상수에 대한 정의를 포함하며, 내용을 %{와 %} 사이에 기술하면 렉스가 생성하는 lex.yy.c의 앞부분에 그대로 삽입된다. 그러나 %{ ... %}의 외부에서 코멘트를 사용할 때는 반드시 하나 이상의 화이트 스페이스(white space)(스페이스, 탭, 줄 바꿈 문자(new line) 등 화면상에는 나타나지 않는 문자)로 행이 시작되어야 한다.

13.2 렉스와 야크의 입력 파일 형식

- 정의 부분은 다음과 같이 구성된다.

```
%{
 /* lex.yy.c의 앞부분에 그대로 복사되는 부분 */
 /* 자료 구조, 변수 및 상수 정의 */
%}
이름1  치환식1
이름2  치환식2
  ⋮      ⋮
이름n  치환식n
```

13.2 렉스와 야크의 입력 파일 형식

- 변환 규칙 부분은 정규 표현과 수행 코드의 쌍으로 구성된다. 변환 규칙 부분의 구성은 다음과 같다.

```
%%
규칙1  수행1
규칙2  수행2
 ⋮      ⋮
규칙n  수행n
```

- 여기서 %%는 정의 부분과 변환 규칙 부분을 구분하는 구분자이다. 그리고 규칙$_n$은 인식하고자 하는 토큰을 나타내는 정규 표현이고, 수행$_n$은 토큰이 인식되었을 때 어휘 분석기가 처리 할 수행 코드이다.

13.2 렉스와 야크의 입력 파일 형식

- **[예제 13-1] 렉스의 입력으로 식별자를 정의하고 식별자가 나타나면 토큰 값 리턴하기**
  - 렉스의 입력으로 식별자를 정의하고, 식별자가 나타나면 식별자에 대한 토큰 값을 리턴하는 입력을 작성해보자. 단, 식별자는 C 언어에 대한 식별자이다.
  - [풀이]
  - Letter [a-zA-Z_]
  - Digit [0-9]
  - %%
  - {Letter}({Letter} | {Digit})* return tident
- **[예제 13-2] 변환 규칙 부분 작성하기**
  - 입력 문자열에서 숫자를 만나면 'found integer number'를 출력하는 변환 규칙 부분을 작성해보자.
  - [풀이]
  - [0-9]+ printf("found integer number₩n");

13.2 렉스와 야크의 입력 파일 형식

- 변환 규칙 부분에서 토큰을 표현하려면 정규 표현으로 패턴을 정의해야 하는데, 정규 표현에는 [표 13-1]과 같은 메타 기호를 사용한다.

표 13-1 메타 기호와 의미

메타 기호	의미
" "	이중 부호 " " 속에 있는 모든 문자는 텍스트 문자로 취급된다.
\	역빗금은 하나의 문자를 에스케이프하는 데 사용된다. 하나의 문자를 텍스트 문자로 취급하고자 할 때 특수 문자 앞에 역빗금을 덧붙임으로써 해결할 수 있다.
[ ]	문자의 종류를 정의하는 데 사용한다. [ ] 속에 사용된 대부분의 기호는 의미를 상실하고 –, ^, \만이 특수한 의미를 가진다.
*	영 번 이상의 반복을 나타낸다.
+	한 번 이상의 반복을 나타낸다.
?	선택을 의미하는 기호로 ? 앞에 있는 문자가 선택적임을 나타낸다.
|	택일을 나타내는 기호이다.
^	첫 문자로 사용할 때는 행의 시작을 의미한다.
$	행의 끝으로만 인식된다.
/	/ 뒤에 있는 정규식과 일치하는 경우에만 / 앞에 있는 정규식에 대응된다.
.	줄바꿈 문자를 제외한 모든 문자를 의미한다.

13.2 렉스와 야크의 입력 파일 형식

- 다음은 메타 기호를 이용하여 자주 사용되는 예이다.
  - a"*"b : 문자열 a*b를 나타낸다. 문자열 a*b는 정규 표현 a*b와 같다.
  - a₩*b : 문자열 a*b를 나타내는 정규 표현이다.
  - [abc] : a, b, c 중의 한 문자를 나타낸다.
  - [a-z] : 문자 a부터 z 중 하나, 즉 소문자를 나타낸다.
  - [0-9] : 숫자 0부터 9 중 하나, 즉 범위를 나타낸다.
  - [^+] : +를 제외한 모든 문자, 즉 여집합을 나타낸다.
  - [₩t₩n] : 공백, 탭, 줄 바꿈 문자 중의 하나를 나타낸다.
  - [a-zA-Z][a-zA-Z0-9]* : 첫 자는 영문자이고 둘째 자부터 영문자나 숫자로 구성된 문자열이다.
  - a+ : 문자 a가 한 번 이상 반복되는 것을 나타낸다.
  - ab | c : ab 혹은 c를 나타낸다.
  - ^ab : 행의 시작에 문자열 ab가 나타나는 경우에만 토큰으로 ab를 처리한다.
  - ab$: 행의 끝에 문자열 ab가 나타나는 경우에만 토큰으로 ab를 처리한다.
  - ab/cd : ab 다음에 cd가 이어서 나타날 때만 ab가 토큰으로 처리된다.

13.2 렉스와 야크의 입력 파일 형식

- 렉스 입력의 마지막인 사용자 부프로그램 부분은 렉스에서 사용될 부프로그램을 정의하는 부분으로 렉스에서 어떤 처리 없이 그대로 렉스의 출력인 lex.yy.c에 복사된다.
- 렉스에서는 여러 정보를 제공하고 복잡한 기능을 수행할 수 있도록 하기 위해 다양한 함수와 전역 변수를 제공한다. 이러한 함수와 전역 변수는 [표 13-2]와 같다.

표 13-2 렉스의 함수와 전역 변수

함수와 변수 종류	이름	의미
전역 변수	yytext	입력 문자열에서 정규 표현에 의해 실제로 매칭된 문자열을 갖는 문자 배열형이다.
	yyleng	매칭된 문자열의 길이를 저장하는 변수이다.
함수	yylex( )	렉스 입력에서 명시한 정규 표현과 일치하는 토큰을 찾을 때까지 한 문자씩 계속 읽는 함수이다.
	yywrap( )	렉스가 입력의 끝을 만났을 때 호출하는 함수로 정상적인 함수 복귀 값은 1이다.
	yymore( )	현재 매칭된 문자열의 끝에 다음에 인식될 문자열을 덧붙이는 함수이다.
입출력 함수	input( )	입력 문자열로부터 다음 문자를 읽는 함수이다.
	output(c)	출력 문자열로 문자 c를 내보내는 함수이다.
	unput(c)	input(c)에 의해 읽히도록 문자 c를 입력 문자열로 되돌려보내는 함수이다.

13.2 렉스와 야크의 입력 파일 형식

- 토큰 값이 필요한 경우에 변수 yytext의 내용을 사용하여 해결할 수 있다.
  - 예를 들어 매칭된 문자열을 출력할 때 다음과 같이 작성한다.
  - [a-z]+ printf("%s", yytext);
  - 이것은 입력에서 소문자로 이뤄진 단어를 찾으면 그 단어를 yytext에 저장하고 그 내용을 그대로 출력하라는 의미이다. 이와 동일한 기능을 하는 렉스의 함수로 ECHO가 있는데, 다음과 같이 작성하면 똑같은 결과를 얻을 수 있다,
  - [a-z]+ ECHO;
- yytext[yyleng-1]이라고 하면 매칭된 문자열의 마지막 문자를 출력한다.
- 렉스의 출력인 lex.yy.c에는 스캐너 역할을 하는 함수 yylex( )가 포함되어 있다. 이 함수는 렉스의 입력에서 명시한 정규 표현과 일치하는 토큰을 찾을 때까지 한 문자씩 계속 읽어 들인다. 그리고 토큰이 매칭되면 해당 정규 표현과 결합된 수행 코드가 실행되고, 함수 yylex( ) 의 복귀 값 return value이 바로 토큰 번호에 해당된다. 또한 yylex( )를 호출한 함수가 토큰 값 을 요구할 때는 외부 변수 yytext의 내용을 복사하여 필요로 하는 함수에 전달한다. 함수 yylex( )는 입력 파일의 끝에 도달할 때까지 토큰을 생성한다. 파일의 끝에 도달하면 yylex( ) 는 함수 yywrap( )를 호출한 후 0으로 복귀한다. 그러므로 모든 토큰을 처리한 후 실행할 일이 더 있는 경우 yywrap( ) 내에서 처리하면 된다.

13.2 렉스와 야크의 입력 파일 형식

▪ 야크의 입력 파일 형식

- 야크의 입력은 문법 규칙과 각 규칙에 대한 수행 코드이며 다음과 같이 선언 부분(declation part), 변환 규칙 부분(translation rules part), 사용자 프로그램 부분(user program part)으로 구성된다.

```
  선언 부분
%%
  변환 규칙 부분
%%
  사용자 프로그램 부분
```

- 각 부분은 %%로 구분되는데, % 기호는 야크의 표현에서 에스케이프(escape) 문자로 사용된다. 선언 부분은 생략할 수 있으며, 만약 사용자 프로그램 부분이 생략되는 경우 두 번째 %% 기호도 생략 가능하다. 선언 부분에서는 문법 규칙의 토큰에 대해 선언하고 변환 규칙 부분과 프로그램 부분에서 사용될 임시 변수에 대해 선언한다.

13.2 렉스와 야크의 입력 파일 형식

- 시작 기호는 다음과 같이 정의한다.
- %start
- %{와 %} 사이에 내용을 기술하며, 어휘 분석기에서 반환되는 모든 토큰에 대한 정의를 %token으로 정의한다. 관례적으로 모든 토큰의 이름은 대문자로 하고 토큰 이외의 다른 이름은 소문자로 한다.
- 변환 규칙 부분에서는 문법의 규칙을 정의한다. 변환 규칙 부분에 대한 표현은 일련의 문법 규칙으로 구성되고, 각 규칙은 왼쪽 부분(left hand side)과 몸체(body)의 형태를 취한다. 여기서 왼쪽 부분은 논터미널의 이름이고, 몸체는 문법 규칙의 오른쪽 부분에 나오는 기호로 이름이나 상수 문자가 될 수 있다. 야크의 각 규칙에 대한 수행 코드는 C 언어로 기술되며, 수행 코드는 각 규칙이 감축될 때 실행된다.
- 사용자 프로그램 부분은 야크를 보조하는 C 루틴으로서 오류 처리 루틴 등이 첨가된다.

13.2 렉스와 야크의 입력 파일 형식

[예제 13-3] 규칙을 야크로 표현하기

- BNF로 표현된 규칙을 야크로 표현해보자.
- <expr> ::= <expr> + <term>
- [풀이]
- 이를 야크로 표현하면 다음과 같다.
- expr : expr ‘+’ term

- 사용자는 각 생성 규칙 및 그 생성 규칙이 파서에 의해 인식되었을 때 호출되는 수행 코드를 작성해야 한다. 생성 규칙에 해당하는 수행 코드를 작성해보자.
- expr : expr ‘+’ term { printf(“addition expr detected ₩n”);};
- 야크에서는 생성 규칙에 있는 문법 기호에 대한 값을 정의할 수 있도록 의사 변수(pseudo variable) 를 제공한다. 변수 $1과 $2는 각각 오른쪽 부분의 첫 번째 기호와 두 번째 기호의 값을 의미 한다. 생성 규칙의 왼쪽 부분에 있는 기호는 의사 변수 $$로 치환문을 통해 다른 문법 규칙에 값을 전달한다.
- 다음은 의사 변수를 사용하여 수행 코드를 기술한 예이다.
- expr : expr ‘+’ term {$$ = $1 + $3}
- | expr ‘-’ term {$$ = $1 - $3}

13.2 렉스와 야크의 입력 파일 형식

- 5장과 6장에서 모호한 문법에 대한 해결 방법을 다루었는데, 6장에서는 모호한 문법을 사 용하여 파싱표에서 충돌이 발생했을 때 그 충돌을 우선순위와 결합 법칙을 이용해서 해결했다. 같은 방법으로 모호한 문법에 대해 야크를 통해 파싱표를 만든다고 해도 충돌이 발생한 다. 이를 해결하기 위해 야크에서는 우선순위를 제공하는데 가장 낮은 우선순위인 터미널 부터 정의하며, 왼쪽 결합 법칙인 경우에는 LEFT로 표기하고 오른쪽 결합 법칙인 경우에는 RIGHT로 표기한다.

- **[예제 13-4] 야크의 입력에서 연산 순서 알아보기**
  - 다음과 같은 야크의 입력을 보고 식 a = c * d - e + f / g에 대한 연산 순서를 알아보자.
  - %right '='
  - %left '+' '-'
  - %left '*' '/'
  - %%
  - expr : expr '=' expr
  - | expr '+' expr
  - | expr '-' expr
  - | expr '*' expr
  - | expr '/' expr

13.2 렉스와 야크의 입력 파일 형식

- [풀이] 식 a = c * d - e + f / g에 대한 야크의 입력을 보면 연산자 우선순위는 = < +, - < *, /의 순이며, =은 오른쪽 결합 법칙을 취하고 +, -, *, /는 왼쪽 결합 법칙을 취한다. 그러므로 괄호를 이용하여 표기하면 다음과 같다.
- (a = (((c * d) - e) + (f / g)))

13.3 플렉스와 바이슨 설치 방법

- 렉스와 야크는 유닉스 시스템의 유틸리티로 개발되었고, 유용성이 확인되면서 선의 솔라리스나 리눅스와 같은 유닉스 계열의 운영체제를 비롯해 마이크로소프트의 윈도 등 다른 운영체제에도 이식되어 사용되었다. 리눅스와 같은 운영체제의 경우 렉스와 야크의 GNU(Gnu's Not Unix) 버전인 플렉스(Flex)와 바이슨(Bison)을 사용할 수 있다. 윈도에서는 PCLex와 PCYacc도 사용되지만 현재 사용하는 방법 중 유닉스 계열 운영체제와 가장 유사하게 사용 할 수 있는 것은 역시 플렉스와 바이슨이다.
- 윈도상에서 시그윈(Cygwin) 시스템과 같이 유닉스 계열 운영체제와 유사한 환경을 사용한다면 처음 설치할 때 플렉스와 바이슨을 선택하여 사용하는 것이 좋다. 만약 시그윈 시스템과 같이 유닉스 계열 운영체제와 유사한 환경 전체의 설치가 필요 없는 경우에는 GNU C 컴파일러(gcc)와 플렉스, 바이슨만을 윈도에 설치하면 된다.
- 다음과 같은 순서로 윈도에 GNU C 컴파일러(gcc)와 플렉스, 바이슨을 설치한다. 단, GNU C 컴파일러, 플렉스, 바이슨의 설치 순서는 바뀌어도 무관하다.

13.3 플렉스와 바이슨 설치 방법

- **① GNU C 컴파일러를 설치한다.**
  - GNU C 컴파일러를 설치하는 방법 중 하나는 무료 C/C++ 프로그램용 통합 개발 환경(IDE)인 블러드셰드(Bloodshed) 사의 Dev-C++를 설치하는 것이다(http://www.bloodshed.net/devcpp.html에서 내려 받는다). 원래 Dev-C++는 GNU C 컴파일러를 기본으로 하여 그 위에 사용자 편의를 위한 통합 개발 환경 인터페이스를 덮어씌운 것이다. Dev-C++를 설치할 때는 설치 폴더에 이르기까지 경로명에 공백이 없어야 한다. 설치가 끝나면 그 디렉터리 아래에 'bin'이라는 하위 디렉터리가 만들어지고, 그 안에 GNU C 컴파일러(gcc.exe)와 C++ 컴파일러(g++.exe)가 설치된다.
- **② 플렉스와 바이슨을 설치한다.**
  - 플렉스는 http://gnuwin32.sourceforge.net/packages/flex.htm에서 내려받아 설치한다. 설치는 매우 간단하다. Dev-C++와 마찬가지로 경로명에 공백이 없도록 해야 하며 보통은 'C:₩GnuWin32'에 설치한다.
  - 바이슨은 http://gnuwin32.sourceforge.net/packages/bison.htm에서 내려받아 같은 방법으로 설치한다. 역시 경로명에 공백이 없도록 해야 하며, 일반적으로 플렉스와 함께 'C:₩GnuWin32'에 설치한다.

13.3 플렉스와 바이슨 설치 방법

- **③ 시스템 속성에서 환경 변수 중 'PATH' 변수의 값에 경로를 추가한다.**
  - GNU C 컴파일러, 플렉스, 바이슨의 설치를 마치면 시스템 속성에서 환경 시스템의 환경 변수 중 'PATH' 변수의 값에 GNU C 컴파일러와 플렉스, 바이슨의 실행 파일이 있는 경로를 추가한다. 예를 들어 'PATH' 변수가 있으면 'C:₩Dev-Cpp₩bin; C:₩GnuWin32₩bin'을 추가하고, 'PATH' 변수가 없다면 새롭게 등록한다.
- **④ 제대로 설치되었는지 확인한다.**
  - GNU C 컴파일러와 플렉스, 바이슨이 제대로 설치되었는지 확인하기 위해 명령 프롬프트 창을 연다. 명령 프롬프트는 [모든 프로그램] → [보조 프로그램] → [명령 프롬프트] 메뉴를 선택하여 실행하거나 실행 창에서 'cmd.exe'를 입력하여 바로 실행시킨다. 이 창에서 'flex-version'을 입력하여 설치된 플렉스의 버전을 확인하고, 'bison-version'을 입력하여 설치된 바이슨의 버전을 확인한다. 마지막으로 'gcc-version'을 입력하여 설치된 GNU C 컴파일러의 버전을 확인한다. 이 과정에서 버전이 모두 나타난다면 문제없이 제대로 깔린 것이다.

13.3 플렉스와 바이슨 설치 방법

- **⑤ 플렉스와 바이슨이 필요로 하는 라이브러리 파일을 GNU C 컴파일러 디렉터리에 복사한다.**
  - 마지막으로 플렉스와 바이슨이 필요로 하는 라이브러리 파일을 GNU C 컴파일러 디렉터리에 복사한다. 플렉스와 바이슨이 설치된 디렉터리인 'C:₩GnuWin32'에는 여러 하위 디렉터리가 있다. 그 중 'lib' 속에 'libfl.a'와 'liby.a'가 포함되어 있는데 이것들은 각각 플렉스와 바이슨의 라이브러리 아카이브(archive) 파일이다. 이것들을 GNU C 컴파일러의 라이브러리 디렉터리인 'C:₩Dev-Cpp₩lib'에 복사한다. 플렉스 입력 파일의 이름을 확장자가 l인 'test.l'이라 하고 바이슨 입력 파일의 이름을 확장자가 y인 'test.y'라고 하면 플렉스와 바이슨을 사용하는 일반적인 형태는 다음과 같다.
  - C:₩Users₩ex > flex test.l
  - C:₩Users₩ex > bison test.y

13.4 플렉스 사용법

- **[예제 13-5] 플렉스를 이용하여 글자 개수, 단어 개수, 문장 개수를 계산하는 어휘 분석기**
  - 플렉스를 이용하여 글자의 개수(빈칸도 하나의 글자로 계산), 단어의 개수, 문장의 개수를 계산하는 어휘 분석기를 만들어보자.
  - [풀이] 1. 플렉스의 입력 파일을 만든다. 입력 파일의 이름은 WordCount.l이라고 하자.

```
 %{
/*
word count
*/
unsigned charCount=0, wordCount=0, lineCount=1;
%}
word [^ ₩t₩n]+
eol ₩n
%%
{word} {wordCount++; charCount+=yyleng; printf("word : %s₩n", yytext);}
{eol} {charCount++; lineCount++;}
. charCount++;
%%
main()
{
yylex();
printf("number of character : %d ₩n", charCount);
printf("number of word : %d ₩n", wordCount);
printf("number of line %d ₩n", lineCount);
}
```

13.4 플렉스 사용법

- 2. 실행 순서
  - 실행하고자 하는 테스트 파일은 다음과 같은 'datafile2.txt'이다.
  - The token descriptions that lex uses are known as regular expressions, extended versions of the familiar patterns used by the grep and egrep commands. A lex lexer is almost always faster than a lexer that you might write in C by hand.
- 맨 먼저 다음을 실행한다.
- C:₩ > flex WordCount.l
- C:₩ > gcc -o WordCount lex.yy.c -lfl
- C:₩ > WordCount < datafile2.txt
- 3. 실행 결과
  - 다음 도스 화면은 실행 순서와 실행 결과를 보여주는데 235개의 문자, 42개의 단어, 1개의 행이 나타난다.

13.4 플렉스 사용법

- 
C:₩Windows₩system32₩cmd.exe
C:₩CPW₩test>WordCount < datafile2.txt
word : the
word : token
word : descriptions
word : that
word : lex
word : uses
word : are
word : known
word : as
word : regular
word : expressions,
word : extended
word : versions
word : of
word : the
word : familiar
word : patterns
word : used
word : by
word : the
word : grep
word : and
word : egrep
word : commands.
word : A
word : lex
word : lexer
word : is
word : almost
word : always
word : faster
word : than
word : a
word : lexer
word : that
word : you
word : might
word : write
word : in
word : C
word : by
word : hand.
number of character : 235
number of word : 42
number of line 1
C:₩CPW₩test>

13.5 바이슨 사용법

- **[예제 13-7] 플렉스와 바이슨을 이용하여 파싱표와 구문 분석기 만들기**
  - 다음 문법에 대해 바이슨을 이용하여 SLR 파싱표와 구문 분석기를 만들어보자. 이 문법은 [예제 6-30]에서 SLR 방법으로, [예제 6-34]에서 CLR 방법으로, [예제 6-38]에서 LALR 방법으로 파싱표를 만들었다.
  - S → L = R
  - S → R
  - L → *R
  - L → id
  - R → L
  - [풀이]
  - 1. 플렉스 입력 파일 작성(파일명 : TT.L)

```
%{
#include "y.tab.h"
%}
letter [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*
%% /*토큰 정의 규칙*/
"=" return(yytext[0]);
"*" return(yytext[0]);
{id} return(ID);
[ ₩n₩t₩b];
```

13.5 바이슨 사용법

- 2. 바이슨 입력 파일 작성(파일명 : TT.Y)

```
%{
%}
%token ID
%%
S:L '=' R
 | R;
L:'*'R
 | ID;
R:L;
%%
main()
{
if(yyparse()==0)
{
printf("The Parsing Complete ₩n");
}
else
{
printf("syntax error ₩n");
}
}
```

13.5 바이슨 사용법

- 3 실행 순서
- C:₩ > flex tt.l /* lex.yy.c 어휘 분석기 생성 과정 */
- C:₩ > bison -yd -v tt.y
- C:₩ > gcc -o tt lex.yy.c y.tab.c -lfl –ly
-

13.5 바이슨 사용법

- 4. 파싱표
-

상태	구문 분석기의 행동				GOTO 함수		
	=	*	id	$	S	R	L
0		s2	s1		3	5	4
1	r4			r4			
2		s2	s1			7	6
3				s8			
4	s9, r5			r5			
5				r2			
6	r5			r5			
7	r3			r3			
8				acc			
9		s2	s1			10	6
10	r1			r1			

13.5 바이슨 사용법

-

상태	구문 분석기의 행동				GOTO 함수		
	=	*	id	$	S	R	L
0		s4	s5		1	3	2
1				acc			
2	s6			r5			
3				r2			
4		s4	s5			7	8
5	r4			r4			
6		s4	s5			9	8
7	r3			r3			
8	r5			r5			
9				r1			