# 1 렉스와 야크

렉스(lex)와 야크(yacc)는 일정한 구조에 따라 입력된 내용을 변환하는 프로그램을 만드는 데 유용하게 쓰이는 툴이다. 입력 파일에서 일정한 패턴을 찾아내는 간단한 텍스트 검색 프로그램에서 소스 프로그램을 최적화된 목적 코드(object code)로 변형하는 C 컴파일러까지 다양한 프로그램을 만드는 데 도움을 준다.

일정한 구조를 가진 입력을 받아들이는 프로그램에서 지속적으로 반복하는 작업은 입력 내용을 의미 있는 단위(unit)로 분해하여 그들 사이의 관련성을 파악하는 일이다. 텍스트 검색 프로그램에서 의미 있는 단위는 찾고자 하는 문자열을 담고 있는 행과 그렇지 않은 행으로 구분된 각각의 행이 될 수 있다. 한편 C 프로그램의 단위는 변수명, 상수, 문자열, 연산자, 기호 등이다. 이렇게 입력을 (보통 토큰이라고 하는) 단위로 나누는 작업을 대개 어휘 분석(lexical analysis) 혹은 줄여서 렉싱(lexing)이라고 한다. 렉스는 몇 개의 토큰을 받아들여 주어진 토큰을 식별할 수 있는 C 루틴(routine)을 생성한다. 이 루틴을 어휘 분석기(lexical analyzer) 또는 렉서(lexer)나 스캐너(scanner)라고 한다. 한편 여러분이 렉스에 기술하는 내용을 일컬어 렉스 명세서(lex specification)라 한다.

렉스가 사용하는 토큰 설명(token descriptions)을 보통 정규 표현식(regular expressions)이라 하며, grep이나 egrep 같은 유닉스 명령어에서 사용하는 패턴을 크게 확장한 버전이다. 렉스는 찾고자 하는 표현식의 개수에 관계없이 렉서가 입력 텍스트를 최대한 빠르게 검색할 수 있는 형태로 이러한 정규 표현식을 바꾸어 준다. 렉스가 생성하는 렉서는 여러분이 직접 C 언어를 이용해서 작성하는 렉서보다 대부분 빠르게 작동한다.1)

입력된 내용이 토큰으로 분해되고 나면, 프로그램은 이제 토큰 사이의 관련성을 확립해야 한다. 예를 들어, C 컴파일러는 프로그램 안에서 표현식(expressions), 문장(statements), 선언문(declarations), 블록(blocks) 그리고 프로시저(procedures) 등을 각기 구별해서 파악해야한다. 이러한 작업을 파싱(parsing)이라 하며, 프로그램이 이해할 수 있는 관련성을 정의하는 규칙을 문법(grammar)이라 한다. 야크는 문법에 대한 간략한 설명을 받아들여, 그 문법을 파싱할 수 있는 C 루틴, 즉 파서(parser)를 생성한다. 야크가 생성한 파서는 문법에 정의된 규칙과 매치(match)하는 일련의 입력 토큰을 자동으로 인식하고, 매치하지 않는 토큰에 대해서는 구문 에러(syntax error)를 발생시킨다. 대개 야크가 생성한 파서는 여러분이 직접 작성하는 파서보다 빠르지 않지만, 파서를 쉽게 작성하고 수정할 수 있다는 장점은 속도가 다소 늦다는 단점을 상쇄하고도 남는다. 더욱이 프로그램이 파싱을 수행하는 동안 소비하는 시간이 문제가 되는 경우는 드물다.

입력 내용을 단위로 나누고 그 단위들 사이의 관련성을 파악하는 작업이 필요하다면, 렉스와 야크를 사용하는 것을 고려해 볼 필요가 있다(검색 프로그램은 너무 간단하기 때문에 파싱을 수행할 필요는 없다. 따라서 렉스는 사용할지 몰라도 야크는 사용할 필요가 없다. 2장에서는 야크는 사용하지 않고 렉스만 이용하여 몇 가지 애플리케이션을 만들어 본다).

일단 여기에서는 여러분의 흥미를 충분히 자극했기를 바랄 뿐이다. 이 장을 렉스와 야크에 대한 완전한 안내서로 삼기보다는, 그 사용법에 대한 적당한 소개를 목적으로 살펴보기 바란다.

1) 역자주 : 적절한 말이 없어서 렉스와 렉서를 영문 발음으로 번역하였다. 따라서 렉스(lex)와 렉서(lexer)를 구별해서 읽기 바란다. 렉스를 부모로 생각하고 렉서를 자식으로 생각하면 이해하는 데 도움이 될 것이다.

## 가장 간단한 렉스 프로그램

다음 렉스 프로그램은 표준 입력(standard input)의 내용을 표준 출력(standard output)으로 그대로 복사한다:

```
%%
.|\n      ECHO;
%%
```

이 프로그램은 유닉스 cat 명령어에 인자(arguments)를 전달하지 않고 실행할 때와 거의 비슷하게 작동한다.

렉스는 입력 파일을 읽어들이는 실제 C 프로그램 코드를 생성하며, 때로는 이 경우처럼 출력 또한 수행하기도 한다.

프로그램의 일부를 만들기 위해서 렉스와 야크를 사용하든, 단순히 프로그래밍 업무를 수월하게 하기 위한 도구를 만들기 위해 사용하든, 일단 렉스와 야크를 사용하는 데 익숙해지고 나면 복잡한 입력을 쉽게 다룰 수 있는 능력 이상의 가치를 체험하게 될 것이다. 렉스와 야크는 또한 다루기 쉬운 코드 베이스(code base)를 제공할 뿐만 아니라, 여러분의 프로그램에 올바른 의미를 부여하는 작업을 좀더 쉽게 만들 것이다.

## 렉스를 이용한 단어 인식

여러 가지 영어 단어를 인식하는 간단한 프로그램을 작성해 보자. 우리는 우선 (명사, 동사 등과 같은) 언어의 각 요소를 식별하는 작업에서 시작하여, 간단한 영문법을 따르는 여러 개의 단어로 이루어진 문장들을 다루는 작업으로 난이도를 서서히 높여 나갈 것이다.

우리가 여기에서 인식하고자 하는 동사들은 다음과 같다.

```
is     am      are    were
was    be      being  been
do     does    did    will
would  should  can    could
has    have    had    go
```

[예제 1-1]은 위와 같은 동사들을 인식하는 데 필요한 간단한 렉스 명세서이다.

[예제 1-1] 단어 인식기 ch1-02.l

```
%{
/*
 * 이 샘플은 (매우) 간단한 인식 과정을 보여준다.
 * 동사/비동사에 대한 인식
 */
%}
%%

[\t ]+    /* 공백은 무시한다. */    ;

is |
am |
are |
were |
was |
be |
being |
been |
do |
does |
did |
will |
would |
should |
can |
could |
has |
have |
had |
go            { printf("%s: is a verb\n", yytext); }
[a-zA-Z]+     { printf("%s: is not a verb\n", yytext); }

.|\n          { ECHO; /* 기본으로 수행되는 동작 */ }
%%

main()
{
     yylex();
}
```

이 프로그램을 컴파일하여 실행한 결과는 다음과 같다. 우리가 직접 타이핑한 부분은 **볼드체(bold)**로 나타내었다.

```
% example1
did I have fun?
did: is a verb
I: is not a verb
have: is a verb
fun: is not a verb
?
^D
%
```

자세한 내용을 이해하기 위해서 우선 첫번째 절을 살펴보도록 하자.

```
%{
/*
 * 이 샘플은 (매우) 간단한 인식 과정을 보여준다.
 * 동사/비동사에 대한 인식
 */
%}
```

첫번째 절은 정의절(definition section)이라 하며 최종 프로그램에 포함하고자 하는 C 프로그램의 내용을 삽입하는 기능을 담당한다. 그러나 최종 파일 안에 반드시 포함되어야 하는 헤더 파일이 존재하는 경우에는 이 절이 단순한 삽입 이상의 중요한 기능을 담당한다. 이 절에 존재하는 C 코드는 "%{"과 "%}"라는 특별한 분리자(delimiter)로 둘러싸여 있다. 렉스는 이 분리자 사이에 있는 내용을 C 파일에 그대로 복사하기 때문에, 여러분은 이 안에 C 코드를 작성해 넣을 수 있다.

이 예에서는 정의절에 단순히 약간의 C 언어의 주석만이 있었지만, 여러분은 이 주석을 특별한 분리자로 둘러싸지 않고 바로 입력할 수는 없는지 궁금할 것이다. 주석이 분리자 바깥에 존재하는 경우에는 렉스가 이를 주석으로 인식하도록 하기 위해 반드시 공백(whitespace)을 이용해 주석을 들여 써야 한다. 이러한 사실을 깜빡 잊고 주석을 들여 쓰지 않으면 렉스가 그 부분을 주석이 아닌 다른 내용으로 인식하기 때문에 예측 불가능한 버그가 발생한다. %% 기호는 정의절의 끝을 의미한다.

다음절은 규칙절(rules section)이다. 각각의 규칙은 두 부분, 즉 공백으로 구분된 패턴(pattern)과 동작(action)으로 이루어진다. 렉스가 생성하는 렉서는 패턴을 인식했을 때 주어진 동작을 수행한다. 이러한 패턴은 유닉스에서 흔히 사용되는 정규 표현식(regular expressions)과 비슷하지만, grep, sed, ed 등에서 사용하는 표현식보다는 좀더 확장된 내용을 담고 있다. 6장에서 정규 표현식과 관련된 규칙을 자세히 설명한다. 이 예제에서 첫번째 규칙은 다음과 같다.

```
[\t ]+          /* 공백은 무시한다. */ ;
```

대괄호 "[ ]"는 그 안에 있는 임의의 문자 중 하나가 패턴으로 사용된다는 것을 가리킨다. 이 예에서는 (탭 키인) "\t"이나 (스페이스 키인) " "를 받아들인다. "+" 기호는 주어진 패턴이 한 번 이상 반복될 수 있다는 사실을 나타낸다. 따라서 위의 패턴은 결국 (탭 키나 스페이스 키로 이루어진) 공백을 의미한다. 이 예에서 규칙의 두번째 부분인 동작(action)은 단순히 세미콜론이며, C 프로그램에서는 아무 일도 수행하지 않는 문장을 의미한다. 이렇게 별다른 명령 없이 세미콜론만 사용하면 단순히 매치된 입력을 무시한

다.

다음 규칙은 "|"(수직바)를 이용하는 동작이다. 이것은 다음 패턴과 동일한 동작을 수행하라는 의미로서, 이 예에서는 모든 동사들에 대해 맨 마지막 동사에 정의된 동작을 수행하도록 한다.[2)]

첫번째 패턴은 다음과 같다.

```
is |
am |
are |
were |
was |
be |
being |
been |
do |
does |
did |
should |
can |
could |
has |
have |
had |
go          { printf("%s: is a verb\n", yytext); }
```

위의 패턴은 리스트 안에 있는 동사를 매치하는 것이다. 동사를 인식하면 C 언어의 **printf** 문장인 동작을 수행한다. **yytext**라는 배열은 방금 패턴에 매치된 텍스트를 저장한다. 여기에서 수행되는 동작은 인식된 동사의 뒤에 ": is a verb\n"이라는 문자열을 붙여 출력하는 것이다.

마지막 두 규칙은 다음과 같다.

2) 패턴 안에서도 수직바를 사용할 수 있다. 예를 들어, foo|bar는 문자열 "foo" 혹은 "bar"에 매치하는 패턴을 의미한다. 이와 같이 패턴 안에서 이용되는 수직바 기호와 구분하기 위해서 동작을 의미하는 수직바에서는 약간의 스페이스를 사용하였다.

```
[a-zA-Z]+{ printf("%s: is not a verb \n", yytext); }
.|\n      { ECHO; /* 기본으로 수행되는 동작 */ }
```

"[a-zA-Z]+"는 흔히 사용되는 패턴이며, 최소한 문자 하나 이상으로 이루어진 알파벳 문자열을 의미한다. "-" 문자가 대괄호 안에서 사용되는 경우에는 특별한 의미가 있으며, "-" 왼쪽에 있는 문자에서 오른쪽에 있는 문자 사이의 범위에 있는 문자들을 가리킨다. 주어진 입력이 이 패턴에 매치하면, 매치된 토큰 뒤에 ": is not a verb\n"라는 문자열을 붙여 출력하는 동작이 수행된다.

앞서 나열된 동사들이 모두 이 패턴([a-zA-Z])에도 매치한다는 사실을 깨닫는 것은 별로 어렵지 않다. 그렇다면 리스트에 있는 동사를 발견했을 때 두 가지 동작을 모두 수행하지 않는 이유가 궁금할 것이다. 또한 "island"는 "is"로 시작하고 있으므로 "island"라는 단어가 발견되었을 때 두 동작이 모두 실행되지 않을까 할 것이다. 이러한 질문에 답하기 위해서는, 렉스에 모호함을 제거하는 규칙이 있다는 사실을 알아야 한다. 렉서가 제대로 동작하도록 하는 두 가지 규칙은 다음과 같다.

1. 렉스 패턴은 주어진 문자나 문자열에 오직 한 번만 매치한다.
2. 렉스는 현재 주어진 입력에서 패턴에 매치하는 문자 또는 문자열 중 가장 길게 매치할 수 있는 것(longest possible match)에 대한 동작을 수행한다. 따라서 "island"와 "is" 모두 매치할 수 있지만, "island"가 "is"를 포함하면서도 더 길기 때문에 이를 주어진 패턴에 매치한 문자열로 처리한다.

위와 같이 렉스가 생성한 렉서에서 패턴 매칭을 수행하는 방식을 생각해 보면, 예제에 나열된 동사만 매치되는 이유를 알 수 있다.

마지막 행은 기본적인 경우이다. 특수 문자인 "."(마침표)는 개행 문자를 제외한 임의의 문자 하나에 매치하고, "\n"은 개행 문자와 매치한다. 특수 동작인 **ECHO**는 구두점(punctuation)이나 다른 문자들을 모두 그대로 복사하기 때문에 매치된 패턴을 그대로 출력한다. 사실 **ECHO**는 기본적으로 수행되는 동작이지만, 여기서는 명시적으로 포함한 것이다. 그 이유는 기본 패턴이 예상하지 못한 입력에 매치한 경우 제대로 작동하지 않고 이상한 결과를 출력하는 상황을 종종 목격했기 때문이다(매치되지 않은 입력에 대해서도 기본적으로 수행되는 동작이 있지만, 잘 작성된 렉서에는 매치 가능한 모든 입력에 대

한 명시적인 규칙이 지정되어 있다).

규칙절의 끝도 마찬가지로 %%로 구분된다.

마지막 절은 사용자 서브루틴절(user subroutine section)이며, 일반적인 C 코드가 작성될 수 있는 곳이다. 렉스는 생성한 C 코드 뒤에 이 절에 있는 내용을 복사한다. 이 예에서는 **main( )** 프로그램을 작성하였다.

```
%%

main()
{
    yylex();
}
```

렉스가 생성한 렉서는 **yylex( )**라는 C 루틴이므로, 메인 함수(**main**) 안에서 호출할 수 있다.3) 각각의 동작에 명시적인 **return**문이 없다면, **yylex( )**는 주어진 입력 전체를 처리할 때까지 리턴 하지 않는다.

이 예제는 두번째이므로 ch1-02.l이라는 파일에 저장하였다. 우리가 사용하는 유닉스 시스템에서 실행 가능한 프로그램을 만들기 위해서 다음과 같은 명령어를 입력하였다.

```
% lex ch1-02.l
% cc lex.yy.c -o first -ll
```

렉스는 렉스 명세서를 lex.yy.c라는 C 소스 파일로 변환하는데, 위 명령어는 이 C 소스 파일을 렉스 라이브러리를 포함하기 위한 -ll 플래그와 함께 컴파일하고 링크하는 것이다. 그 다음, 이 실행 프로그램이 앞에서 본 것처럼 제대로 작동하는지 확인하기 위해서 직접 실행해 보았다. 이렇게 간단한 렉스 명세서를 이용하였을 뿐인데도 인식하고자 했던 동사들이 실제로 인식되고 있음을 직접 실행하여 확인해 보기 바란다.

3) 사실 렉스 라이브러리는 이와 같은 메인 루틴을 기본적으로 포함하고 있기 때문에 이 메인 루틴을 생략해도 좋다.

첫번째 예제를 끝마쳤으므로, 이제 약간 '멋을 부려 보자. 두번째 예제인 [예제 1-2]는 다른 언어 요소들도 인식하도록 렉서를 좀더 확장하고 있다.

[예제 1-2] 언어의 여러 요소를 인식하는 렉스의 예 ch1-03.l

```
%{
/*
 * 다른 언어 요소에 대한 인식을 추가함으로써 첫번째 예제를 확장하였다.
 */
%}
%%
[\t ]+    /* 공백은 무시한다. */    ;
is |
am |
are |
were |
was |
be |
being |
been |
do |
does |
did |
will |
would |
should |
can |
could |
has |
have |
had |
go            { printf("%s: is a verb\n", yytext); }
very |
simply |
gently |
quietly |
calmly |
angrily       { printf("%s: is an adverb\n", yytext); }
to |
from |
behind |
above |
below |
between |
below         { printf("%s: is a preposition\n", yytext); }
if |
then |
and |
but |
or            { printf("%s: is a conjunction\n", yytext); }
their |
my |
your |
his |
her |
its           { printf("%s: is an adjective\n", yytext); }
I |
you |
he |
she |
we |
they          { printf("%s: is a pronoun\n", yytext); }
[a-zA-Z]+ {
              printf("%s:  don't recognize, might be a noun\n", yytext);
         }
 .|\n    { ECHO; /* 기본으로 수행되는 동작 */ }
%%
main()
{
     yylex();
}
```

## 기호 테이블

두번째 예제도 실제로는 크게 다르지 않다. 앞에서 보았던 예보다 약간 더 많은 단어를 나열했을 뿐이며, 원리상으로는 이 예제 안에 얼마든지 많은 단어를 더 집어넣을 수도 있다. 그렇지만 렉스가 실행 중에 인식하는 단어를 저장하기 위한 내부 테이블을 동적으로 작성하도록 하면, 새로운 단어를 추가하고 싶을 때마다 굳이 렉스 프로그램을 수정하여 다시 컴파일하지 않아도 되기 때문에 더 편리할 것이다. 다음 예제에서는 바로 이러한 기능을 추가할 것이다. 즉, 렉서가 실행하고 있는 동안 입력 파일에서 단어들을 읽어 들여, 인식하고자 하는 언어의 요소를 동적으로 선언하도록 하는 것이다. 선언 행은 언어의 각 요소명으로 시작하며, 선언하고자 하는 단어들이 그 뒤를 따른다. 예를 들어, 다음은 각각 명사 네 개와 동사 세 개를 선언하는 행이다.

```
noun dog cat horse cow
verb chew eat lick
```

이러한 단어들로 이루어진 테이블을 간단히 기호 테이블(symbol table)이라 하며, 렉스나 야크 애플리케이션에서 흔히 사용되는 구조이다. 예를 들어, C 컴파일러는 변수, 구조체명, 레이블, 열거 태그(enumeration tag) 그리고 프로그램에서 사용하는 다른 모든 이름들을 기호 테이블 안에 저장한다. 각 이름은 자신을 기술하는 정보와 함께 저장된다. C 컴파일러에서 이러한 정보는 기호의 타입, 선언의 범위, 변수의 타입 등이다. 이 예에서는 언어의 요소가 이름에 대한 정보가 된다.

기호 테이블에 내용을 추가하면 실질적으로는 렉서 자체가 변화한다. 우리는 렉서가 매치해야 하는 각 단어를 개별적인 패턴으로 표현하기보다, 모든 단어에 매치할 수 있는 패턴 하나를 정의하여, 매치된 입력이 언어에서 어떤 요소를 의미하는지 파악하기 위해 기호 테이블을 참조하였다. (명사, 동사 등과 같은) 언어 요소명은 선언문에서 맨 앞을 구성하므로 사실상 '예약어(reserved words)'에 해당한다. 그러나 각각의 예약어에 대한 독립적인 렉스 패턴은 여전히 존재한다. 기호 테이블을 관리하는 루틴도 작성해야 하는데, 이 경우 **add_word( )** 루틴은 새로운 단어를 기호 테이블 안에 집어넣는 기능을 수행하고, **lookup_word( )** 루틴은 이미 입력된 단어를 검색하는 기능을 수행한다.

프로그램 코드 안에서는 **state**라는 변수를 선언하였는데, 이 변수에 LOOKUP이라는 값이 있으면 단어를 검색하고 있는 상태를 나타내고, 그밖에 다른 값이 저장되어 있으면

해당 값이 우리가 선언하는 단어의 종류를 나타낸다. 행이 언어의 요소명으로 시작하는 경우에는 **state**를 그 이름이 대표하는 단어를 선언하는 것으로 설정하고, \n을 만나면 **state**를 다시 기본 검색 상태로 되돌린다.

[예제 1-3]은 정의절을 보여주고 있다.

[예제 1-3] 기호 테이블을 이용하는 렉서(세 부분 중 첫번째) ch1-04.l

```
%{
/*
 * 기호 테이블을 이용하는 단어 인식기
 */
enum {
     LOOKUP = 0, /* 기본 값 - 정의하기보다는 우선 찾아본다. */
     VERB,
     ADJ,
     ADV,
     NOUN,
     PREP,
     PRON,
     CONJ
};
int state;
int add_word(int type, char *word);
int lookup_word(char *word);
%}
```

테이블에 각 단어의 종류를 기록하고, 변수 **state**의 값을 지정할 수 있도록 enum을 정의하였다. 이 열거형(enumerated type)은 **state** 변수를 통해 우리가 정의하는 내용이 무엇인지 확인할 때는 물론, 각 단어가 어떤 타입인지 확인하는 경우에도 사용된다. 한편 기호 테이블을 위한 루틴도 선언하였다.

[예제 1-4]는 규칙절을 보여준다.

[예제 1-4] 기호 테이블을 이용하는 렉서(세 부분 중 두 번째) ch1-04.l

```
%%
\n        { state = LOOKUP; }    /* 행의 끝, 기본 상태로 되돌아간다. */
          /* 특정 행이 예약된 요소명으로 시작할 때마다 */
          /* 해당 타입의 단어를 정의하기 시작한다. */
^verb     { state = VERB; }
^adj      { state = ADJ; }
^adv      { state = ADV; }
^noun     { state = NOUN; }
^prep     { state = PREP; }
^pron     { state = PRON; }
^conj     { state = CONJ; }

[a-zA-Z]+ {
              /* 일반 단어의 경우, 정의하거나 찾아본다. */
          if(state != LOOKUP) {
              /* 현재 단어를 정의한다. */
              add_word(state, yytext);
            } else {
              switch(lookup_word(yytext)) {
              case VERB: printf("%s: verb\n", yytext); break;
              case ADJ: printf("%s: adjective\n", yytext); break;
              case ADV: printf("%s: adverb\n", yytext); break;
              case NOUN: printf("%s: noun\n", yytext); break;
              case PREP: printf("%s: preposition\n", yytext); break;
              case PRON: printf("%s: pronoun\n", yytext); break;
              case CONJ: printf("%s: conjunction\n", yytext); break;
              default:
                     printf("%s:  don't recognize\n", yytext);
                     break;
              }
           }
```

```
                }
  .     /* 그 밖에 다른 것은 무시한다. */ ;
%%
```

단어를 선언하는 경우, 첫번째 그룹의 규칙은 현재 선언되고 있는 언어 요소를 나타내는 타입으로 현재 상태를 설정한다(맨 앞에 있는 캐럿 "^" 기호는 패턴이 입력 행의 맨 앞부분에서 매치하도록 한다). 우리는 각 행이 시작할 때마다 state의 값을 **LOOKUP**으로 설정하여 단어를 입력할 때마다 단어 테이블이 제대로 동작하는지 점검하도록 하였다. 매치한 패턴이 "[a-zA-Z]+"일 때 state가 **LOOKUP**으로 설정되어 있으면 **lookup_word( )**를 이용해서 단어를 검색하고, 단어가 발견되면 그 단어의 타입을 출력한다. 그러나 state에 다른 값이 설정되어 있으면 단순히 **add_word( )**를 이용해서 그 단어를 정의한다.

[예제 1-5]에 있는 사용자 서브루틴절은 동일한 골격의 **main( )** 루틴과 두 보조함수를 담고 있다.

[예제 1-5] 기호 테이블을 이용하는 렉서(세 부분 중 세 번째) ch1-04.l

```
main()
{
     yylex();
}
/* 단어와 타입의 연결 리스트를 정의한다. */
struct word {
     char *word_name;
     int word_type;
     struct word *next;
};
struct word *word_list; /* 단어 리스트의 첫번째 요소 */
extern void *malloc();
int
add_word(int type, char *word)
{
     struct word *wp;
     if(lookup_word(word) != LOOKUP) {
         printf("!!! warning: word %s already defined \n", word);
         return 0;
     }

     /* 단어가 없으므로, 새로운 엔트리를 할당하고 그것을 리스트에 연결한다. */
     wp = (struct word *) malloc(sizeof(struct word));
     wp->next = word_list;
     /* 단어 자체도 복사해야 한다. */

     wp->word_name = (char *) malloc(strlen(word)+1);
     strcpy(wp->word_name, word);
     wp->word_type = type;
     word_list = wp;
     return 1;     /* 정상적으로 작동했다. */
}
int
lookup_word(char *word)
{
     struct word *wp = word_list;
     /* 단어를 찾기 위해서 리스트를 검색한다. */
       for(; wp; wp = wp->next) {
         if(strcmp(wp->word_name, word) == 0)
             return wp->word_type;
     }
     return LOOKUP;    /* 발견되지 않았다. */
}
```

마지막 두 함수는 단어로 구성된 연결 리스트(linked list)를 생성하고 검색을 수행한다. 하지만 각각의 단어를 찾기 위해서 리스트 전체를 검색해야 하기 때문에 많은 단어가 존재하는 경우에는 속도가 저하될 것이다. 실제 소프트웨어 제품을 제작하는 경우에는 해시 테이블(hash table)등 좀더 복잡하더라도 성능이 빠른 방법을 이용할 것이다. 이 예는 비록 속도가 느리긴 하지만 필요한 기능은 충실하게 수행한다.

다음은 방금 전 예제를 실행하여 얻은 세션(session)의 한 예이다.

```
verb is am are was were be being been do
is
is: verb
noun dog cat horse cow
verb chew eat lick
verb run stand sleep
dog run
dog: noun
run: verb
chew eat sleep cow horse
chew: verb
eat: verb
sleep: verb
cow: noun
horse: noun
verb talk
talk
talk: verb
```

이 예제를 충분히 이해할 때까지 직접 실행해 볼 것을 강력히 권한다.

## 문법

몇몇 애플리케이션에서는 우리가 지금까지 살펴본 간단한 단어 검색 기능만으로도 충분할 것이다. 그러나 다른 애플리케이션에서는 특정한 일련의 토큰을 인식하여 적절한 동작을 수행해야 하는 경우가 있을 것이다. 전통적으로 이와 같은 동작들로 이루어진 집합에 대해 기술해 놓은 것을 문법이라고 한다. 이 용어는 특히 우리가 지금 사용하고 있는 예제에 특히 적합하다. 평범한 문장을 인식하고자 한다고 가정하자.

다음은 간단한 문장의 형태를 나타낸 것이다.

```
noun verb
noun verb noun
```

이 시점에서 문법을 기술하기 위한 표기법을 정해 두는 것이 편리하다. 오른쪽 화살표 "→"는 특정 토큰이 새로운 기호로 대체될 수 있다는 사실을 의미한다.[4] 예를 들어, 다음과 같이 표기했다고 하자.

```
subject → noun | pronoun
```

위의 표기는 새로운 기호인 "subject(주어)"가 명사(noun)나 혹은 대명사(pronoun)를 대체할 수 있다는 사실을 뜻한다. 여기서는 원래 존재하던 기호의 의미를 바꾸기보다는 앞에서 정의한 기본 기호를 이용해서 새로운 기호를 정의하였다. 좀더 예를 들자면, 목적어(object)는 다음과 같이 정의할 수 있다.

```
object → noun
```

이제 영어 문법에 완전히 매치하지는 않지만, 다음과 같이 한 문장을 정의할 수 있다.

```
sentence → subject verb object
```

사실 좀더 다양한 종류의 문장에 들어맞도록 위와 같이 간단한 문장의 정의를 확장할 수 있다. 하지만 일단은 우리가 필요한 테스트를 대화식으로 수행할 수 있도록 하는 야크 문법을 정의하는 데 초점을 두기로 한다. 야크 문법을 소개하기 이전에, 새로운 파서에 유용한 값을 전달하기 위해서는 어휘 분석기를 약간 수정해야 한다.

4) 앞서 렉서가 리턴하는 기호를 지칭하기 위해서 "토큰"이라는 용어를 사용했는데, 화살표 왼쪽에 위치한 기호들은 렉서가 리턴하는 값이 아니기 때문에, 여기서는 토큰이 아니라 기호라는 용어를 사용하고 있음에 유의하기 바란다. 모든 토큰은 기호이지만, 모든 기호가 토큰이 되지는 않는다.

## 파서와 렉서의 의사 소통

렉스 스캐너와 야크 파서를 함께 이용하는 경우에는 파서가 좀더 높은 수준의 루틴에 해당한다. 야크 파서는 입력에서 토큰을 얻고자 할 때마다 렉서의 **yylex( )** 함수를 호출한다. 그러면 렉서는 토큰을 인식하기 위해서 입력을 전체적으로 스캔한다. 이때 파서가 관심을 가질 만한 토큰을 만나면, 그 토큰 코드를 **yylex( )**의 값으로 하여 파서에 리턴한다.

그러나 파서가 모든 종류의 토큰에 관심을 두지는 않는다. 예를 들어, 대부분의 프로그래밍 언어에서는 파서가 주석이나 공백과 같은 토큰에 대해 관심을 두지 않는다. 따라서 렉서에서 이처럼 파서가 무시해버리는 토큰을 발견했을 경우엔 구태여 이를 파서에게 리턴하지 않고 계속해서 다음 토큰을 검색한다.

렉서와 파서는 토큰 코드에 대해 미리 의견 일치를 보아야 한다. 우리는 토큰 코드를 야크에서 정의하도록 함으로써 이 문제를 해결하였다. 이 문법에 존재하는 토큰들은 언어를 구성하는 요소들인 **명사(NOUN), 대명사(PRONOUN), 동사(VERB), 부사(ADVERB), 형용사(ADJECTIVE), 전치사(PREPOSITION)** 그리고 **접속사(CONJUNCTION)**이다. 야크는 전처리기 #define을 이용해서 이러한 언어의 각 요소를 작은 정수 값으로 정의한다. 이 예에서 사용되는 값은 다음과 같다.

```
# define NOUN 257
# define PRONOUN 258
# define VERB 259
# define ADVERB 260
# define ADJECTIVE 261
# define PREPOSITION 262
# define CONJUNCTION 263
```

입력이 논리적으로 끝나는 경우 항상 토큰 코드 0이 리턴된다. 야크에서는 0에 대한 기호를 정의하지 않지만, 원한다면 여러분이 직접 기호를 정의할 수도 있다.

야크는 모든 토큰의 정의를 담고 있는 C 헤더 파일을 작성할 수도 있다. 즉 유닉스 시스템에서는 y.tab.h, MS-DOS에서는 ytab.h 혹은 yytab.h인 이 파일을 렉서 안에 포함하면, 정의된 프리프로세서 기호를 렉서의 동작 코드가 이용한다.

## 언어 요소 분석기

[예제 1-6]은 새로운 렉서의 선언절과 규칙절을 보여주고 있다.

[예제 1-6] 파서에서 호출하게 될 렉서 ch1-05.l

```
%{
/*
 * 이제 상위 수준에 있는 파서가 사용할 어휘 분석기를 만든다.
 */
#include "y.tab.h"     /* 파서에서 정의한 토큰 코드 */
#define  LOOKUP 0      /* 기본 값 - 정의된 단어 타입이 아니다. */
int state;
%}
%%
\n        { state = LOOKUP; }
\.\n      {   state = LOOKUP;
              return 0;     /* 문장의 끝 */
          }
^verb     { state = VERB; }
^adj      { state = ADJECTIVE; }
^adv      { state = ADVERB; }
^noun     { state = NOUN; }
^prep     { state = PREPOSITION; }
^pron     { state = PRONOUN; }
^conj     { state = CONJUNCTION; }
[a-zA-Z]+{
           if(state != LOOKUP) {
              add_word(state, yytext);
           } else {
              switch(lookup_word(yytext)) {
              case VERB:
                return(VERB);
              case ADJECTIVE:
                return(ADJECTIVE);
```

```
                    case ADVERB:
                      return(ADVERB);
                    case NOUN:
                      return(NOUN);
                    case PREPOSITION:
                      return(PREPOSITION);
                    case PRONOUN:
                      return(PRONOUN);
                    case CONJUNCTION:
                      return(CONJUNCTION);
                    default:
                      printf("%s:  don't recognize\n", yytext);
                      /* 리턴하지 않고 그냥 무시한다. */
                    }
                 }
              }
\.    ;
%%
... 앞과 동일한 add_word() 와 lookup_word() 함수...
```

여기에는 몇 가지 중요한 차이점이 존재한다. 렉서 안에서 언어 요소를 지칭하기 위해 사용되는 이름들이 파서에 정의된 토큰 이름을 따르도록 바꾸었다. 또한 인식된 단어의 토큰 코드를 파서에 전달하기 위해서 **return** 문을 추가하였다. 하지만 파서가 렉서 안에 새로운 단어를 정의하는 토큰에 대해서는 관심이 없으므로, 이 경우 **return** 문을 넣지 않았다.

이러한 리턴 문장은 **yylex( )**가 프로그램 안에서 정의된 루틴처럼 작동한다는 사실을 보여준다. 파서가 **yylex( )**를 호출할 때마다, 이 함수는 앞에서 처리한 부분 바로 다음부터 처리를 재개한다. 이로인해 우리가 입력 스트림에 대해 점진적으로 조사를 하거나 연산을 수행할 수 있는 것이다. 첫번째 프로그램에서는 이러한 기능이 필요하지 않았지만, 커다란 프로그램 안에서 렉서를 사용하는 경우 그 유용함은 점점 더 뚜렷해진다.

문장의 끝을 표시하는 규칙도 추가하였다.

```
\.\n {    state = LOOKUP;
          return 0; /* 문장의 끝 */
     }
```

마침표 바로 앞에 있는 백슬래시는 마침표가 정규 표현식의 특수 패턴이 아닌 마침표 기호 그 자체라는 사실을 나타내는 역할을 한다. 따라서 이 규칙은 마침표 바로 다음에 개행이 시작되는 경우와 매치한다. 한편 파서에서 기본으로 제공되는 **main( )** 루틴을 사용하도록 어휘 분석기에서 이 루틴을 생략했다.

## 야크 파서

마지막으로 [예제 1-7]은 야크 문법에 대한 초안을 소개하고 있다.

[예제 1-7] 간단한 야크 문장 파서 ch1-05.y

```
%{
/*
 * 영어 문장을 인식하는 간단한 문법을 위한 렉서
 */
#include <stdio.h>
%}
%token NOUN PRONOUN VERB ADVERB ADJECTIVE PREPOSITION CONJUNCTION
%%
sentence: subject VERB object{ printf("Sentence is valid.\n"); }
     ;
subject: NOUN
     |    PRONOUN
     ;
object:       NOUN
     ;
```

```
%%
extern FILE *yyin;
main()
{
     do
       {
          yyparse();
     }
       while(!feof(yyin));
}
yyerror(s)
char *s;
{
   fprintf(stderr, "%s\n", s);
}
```

야크가 생성한 파서의 구조는 렉스가 생성한 렉서와 비슷하다. 첫번째 절인 정의절은 "%{"과 "%}"로 둘러싸인 리터럴 코드 블록(literal code block)을 담고 있다. 우리는 이 블록을 C 언어의 주석(렉스의 경우와 마찬가지로, 적어도 정의절에서는 C 언어의 주석이 C 코드 블록 내부에 존재한다)과 헤더 파일(여기서는 stdio.h) 한 개를 위해 사용하였다.

다음으로, 어휘 분석기가 전달받고자 하는 토큰들에 대한 정의가 존재한다. 이 예에서는 언어를 구성하는 여덟 가지 요소가 이 토큰에 해당한다. 토큰의 이름은 코드를 읽는 다른 사람들에게 명확한 의미를 전달하도록 잘 선택되어야 하지만, 야크 고유의 의미는 없다. 또한 야크에서는 C 언어에서 사용할 수 있는 식별자(identifier)를 모두 사용할 수 있지만, 토큰 이름은 모두 대문자로 작성하고 그밖에 다른 이름은 대부분 혹은 모두 소문자로 작성하는 것이 관례이다.

첫번째 %%는 규칙절이 시작되고 있음을 의미한다. 두 번째 %%는 규칙절이 끝나고 이제 사용자 서브루틴절이 시작되었다는 것을 가리킨다. 가장 중요한 서브루틴은 렉서의 입력 파일이 끝에 다다를 때까지 **yyparse( )**를 반복적으로 호출하는 **main( )** 루틴이다. **yyparse( )** 루틴은 야크로 생성한 파서이기 때문에, 이 메인 루틴은 입력이 바닥날 때까지 계속 문장을 파싱한다(렉서는 행의 끝에서 마침표를 발견하면 토큰 0을 리턴한다. 이는 현재 주어진 입력에 대한 파싱을 완료했다는 것을 파서에 알리는 신호이다).

## 규칙절

규칙절은 실질적인 문법의 내용들을 제품 규칙(production rules) 혹은 단순히 규칙들의 집합으로서 정의한다(어떤 사람들은 그냥 제품이라고도 한다). 각 규칙은 ":" 연산자 왼쪽에 있는 이름, 오른쪽에 있는 기호 리스트와 동작 코드 그리고 규칙의 끝을 의미하는 세미콜론으로 이루어진다. 기본적으로 첫번째 규칙이 최상위 수준에 해당한다. 즉 파서는 이 규칙, 혹은 일반적으로 말하면 이 초기 규칙 안에 포함되어 있는 여러 가지 규칙들에 대해 매치하는 토큰을 우선하여 검색한다. 규칙의 오른쪽에 있는 표현식은 0개 혹은 여러 개의 이름으로 이루어진 리스트이다. 간단하고 전형적인 규칙은 **NOUN**으로 정의되는 **object** 규칙처럼 오른쪽에 기호가 한 개 있는 것이다. 이 규칙의 왼쪽에 있는 기호는 이후 다른 규칙에서 토큰으로 사용될 수 있다. 따라서, 이렇게 간단한 규칙에서부터 시작해서 복잡한 문법을 창조할 수 있다.

예제에서 사용하는 문법에서는 특수 문자 "|"를 사용하고 있는데, 이것은 현재 규칙이 바로 이전 규칙의 왼쪽에 있는 기호를 공유한다는 의미이다. "|"는 흔히 "or(혹은)"라고 읽는데, 여기서 예로 든 문법에서는 주어(subject)가 **NOUN** 혹은 **PRONOUN**이 될 수 있음을 의미한다. 규칙의 동작 부분은 "{"로 시작해서 "}"로 끝나는 C 블록으로 이루어진다. 파서는 규칙이 매치되면, 즉시 규칙의 맨 끝에 나타낸 동작을 수행한다. 우리가 사용하고 있는 **sentence** 규칙은 한 문장이 성공적으로 파싱되었다고 보고하는 동작을 수행한다. **sentence**가 최상위 수준의 기호이기 때문에 입력 전체가 **sentence** 하나에 매치해야 한다. 파서는 렉서가 (여기서는 main 프로그램에 해당하는) 입력의 맨 끝 부분을 만나게 될 때 자신을 호출한 함수로 리턴한다. 그 이후에 **yyparse( )**를 다시 호출하면 현재 상태를 재 초기화하고 다시 처리하기 시작한다. 예에서는 파서가 "subject VERB object"와 같은 입력 토큰 리스트를 만나면 메시지를 출력한다. 만약 파서가 "subject subject"과 같이 규칙에 맞지 않는 토큰 만나게 된다면 어떤 일이 발생하겠는가? 파서는 우리가 사용자 서브루틴절에 제공하는 **yyerror( )**를 호출한 다음, 특수 규칙인 **error**를 인식한다. 만약 파서가 나머지 부분에 대한 파싱을 계속해서 수행할 수 있다면 에러 복구 코드를 제공함으로써 파서가 다시 정상적인 상태로 되돌아오도록 할 수 있다. 만약 에러 복구가 실패하거나 이 경우처럼 에러 복구 코드가 없다면, **yyparse( )**는 에러 발견 후 자신을 호출한 곳으로 리턴한다.

사용자 서브루틴절에 해당하는 세번째와 마지막 절은 두번째 %% 이후에 시작된다. 이 절에는 어떤 C 코드도 포함할 수 있는데, 이 내용은 생성되는 파서 코드 안으로 그대로 복사된다. 이 예에서는 야크가 생성한 파서가 렉스가 생성한 렉서를 이용하여 컴파일 할 때 반드시 필요한 함수인 **main( )**과 **yyerror( )**만을 포함하였다. 메인 루틴은 렉스 입력 파일인 **yyin**에서 파일의 끝을 만나기 전까지 계속해서 파서를 호출한다. 마지막으로 필요한 루틴은 렉서가 제공하는 함수인 **yylex( )**이다.

이 장의 마지막 예제인 [예제 1-8]에서는 앞에서 정의한 문법을 확장하여, 아직 완전하지는 않지만 더 풍부한 내용의 문장을 인식할 수 있도록 했다. 우리는 여러분이 이 예제를 상세히 살펴볼 것을 권한다. 여러분은 아마도 영어라는 언어를 모호하지 않은 방식으로 설명한다는 것이 얼마나 어려운 일인지 깨닫게 될 것이다.

[예제 1-8] 확장된 영어 파서 ch1-06.y

```
%{
#include <stdio.h>
%}
%token NOUN PRONOUN VERB ADVERB ADJECTIVE PREPOSITION CONJUNCTION

%%
sentence: simple_sentence   { printf("Parsed a simple sentence.\n"); }
     | compound_sentence { printf("Parsed a compound sentence.\n"); }
     ;
simple_sentence: subject verb object
     |   subject verb object prep_phrase
     ;
compound_sentence: simple_sentence CONJUNCTION simple_sentence
     |   compound_sentence CONJUNCTION simple_sentence
     ;
subject: NOUN
     |   PRONOUN
     |   ADJECTIVE subject
     ;
verb:        VERB
```

```
        |    ADVERB VERB
        |    verb VERB
        ;

object:       NOUN
        |    ADJECTIVE object
        ;
prep_phrase:  PREPOSITION NOUN
        ;
%%

extern FILE *yyin;
main()
{
     do
       {
          yyparse();
     }
       while(!feof(yyin));
}

yyerror(s)
char *s;
{
   fprintf(stderr, "%s\n", s);
}
```

우리는 초등학교 영어 수업에서 사용할 만한 전통적인 문법 형식을 이용해 **sentence** 규칙을 확장했다. 즉 문장은 간단한 문장(simple sentence)이거나 아니면 등위 접속사로 연결된 여러 개의 독립적인 절(clauses)로 이루어진 복합문이 될 수 있다. 우리가 여기서 사용하는 어휘 분석기는 "and" 혹은 "but"과 같은 등위 접속사와 "if"와 같은 종속 접속사를 구별하지 않는다.

또한 이 문법에 재귀법(recursion)을 도입하였다. 한 규칙이 직접 혹은 간접적인 방식으로 자기 자신에게 적용되는 것을 의미하는 재귀법은 문법을 기술하는 매우 강력한 도구이기 때문에, 우리는 이 책에서 작성하는 거의 모든 야크 문법에서 이 기법을 사용하였다. 여기서는 **compound_sentence**, **verb** 규칙에서 재귀를 이용하고 있다. **compound_sentence** 에서는 복합문이 접속사로 연결된 두 개 이상의 간단한 문장으로 이루어진다고 정의한다. 우선적으로 가능한 매치는 다음과 같다.

```
simple_sentence CONJUNCTION simple_sentence
```

이것은 "두 개의 절(clauses)"을 정의하고 있는 반면, 다음 예는 "두 개 이상의 절로 이루어지는 경우"를 정의하고 있다.

```
compound_sentence CONJUNCTION simple_sentence
```

재귀와 관련된 내용은 나중에 더욱 자세하게 다룰 것이다.

여기에서 정의한 영어 문법이 그다지 유용하지는 않지만, 렉스를 이용해 단어를 검색하고, 야크를 이용해 검색된 단어들 사이의 관련성을 파악하는 기술 자체는 나중에 더 실질적인 애플리케이션을 작성할 때 이용하는 기술과 크게 다를 바가 없다. 예를 들어, 다음과 같은 C 문장이 있다고 가정하자.

```
if( a == b ) break; else func(&a);v
```

컴파일러는 렉스를 이용해서 **if**, **(**, **a**, **==** 등의 토큰을 인식한 다음, 야크를 이용해서 "a == b"가 **if** 문의 표현부(expression part)에 해당하고, **break**는 그 표현식이 "참"인 경우에 해당하지만 함수 호출은 "거짓"인 경우에 해당한다는 사실을 파악한다.

## 렉스와 야크의 실행

이러한 도구들을 우리가 사용하는 시스템 안에서 실제로 어떻게 만들었는지 설명하는 것으로 결론을 대신한다.

이 장에 있는 다양한 렉서들을 ch1-N.l이라 했는데, 이때 N은 정수 값으로 렉스 명세서 예제의 번호에 해당한다. 이와 마찬가지로 파서는 ch1-M.y라고 했으며, 이때 M도 마찬가지로 예제 번호를 의미한다. 그 다음으로, 출력을 확인하기 위해 다음과 같은 유닉스 명령을 이용하였다.

```
% lex ch1-n.l
% yacc -d ch1-m.y
% cc -c lex.yy.c y.tab.c
% cc -o example-m.n lex.yy.o y.tab.o -ll
```

첫번째 행은 주어진 렉스 명세서에 대해서 렉스를 실행한 것으로, 렉서의 C 코드를 담고 있는 lex.yy.c라는 파일을 생성한다. 두번째 행에서는 야크를 실행하여 y.tab.c와 y.tab.h를 생성한다(y.tab.h는 -d 옵션 스위치를 제공함으로써 생성되며, 토큰에 대한 정의를 담고 있는 헤더 파일이다). 그 다음 행은 두 C 파일을 동시에 컴파일하는 것이다. 그리고 마지막 행은 컴파일 결과로 생성된 두 목적 파일을 링크하며, 유닉스 시스템에서는 보통 /usr/lib/libl.a에 있는 렉스 라이브러리 파일인 libl.a를 이용하기 위해 -ll 옵션을 이용한다. AT&T 렉스와 야크를 사용하지 않고 다른 제품을 사용하는 경우, 서브루틴과 명령어의 이름을 바꿀 필요는 있겠지만 다른 것은 거의 바꾸지 않아도 될 것이다(특히 버클리 렉스와 야크의 경우에는 lex와 yacc라는 이름을 byacc와 flex라는 이름으로 바꾸고 링커를 위한 플래그인 -ll를 생략하기만 하면 된다). 하지만 렉스와 야크의 종류에 따라서 크고 작은 차이가 많이 존재하기 때문에 언제나 이렇게 간단한 것은 아니다. 예를 들어, 만약 야크 대신 그누 바이손(GNU bison)을 사용한다면, C 코드가 생성될 때 ch1-M.tab.c와 ch1-M.tab.h라는 이름이 사용된다. MS-DOS 같이 파일명에 대해 보다 엄격한 제약이 주어지는 시스템에서는 이러한 이름들이 (보통 ytab.c와 ytab.b와 같이) 다른 이름으로 대체되어야 한다. 이렇게 다양한 렉스와 야크의 세세한 차이점들에 대해서는 '부록 A'를 참조하기 바란다.

## 렉스와 손수 작성한 렉서의 비교

어떤 사람들은 C 언어를 이용하여 손수 렉서를 작성하는 것이 별로 어렵지 않기 때문에 굳이 렉스를 배워야 할 까닭을 모르겠다고 말하기도 한다. 물론 사실일수도 있고 그렇지 않을 수도 있다. [예제 1-9]는 주석이나 공백은 무시하고, 간단한 명령어, 숫자, 문자열 그리고 개행을 처리하기에 적합한 렉서를 C 언어로 직접 만든 것이다. [예제 1-10]은 동일한 프로그램을 렉스로 만든 것이다. 렉스로 만든 버전이 C 언어로 손수 만든 것보다 3배 가량 길다. 경험적으로 프로그램에 존재하는 버그의 수가 대략 프로그램의 길이에 비례한다고 보았을 때, 이렇게 C 언어로 작성된 렉서를 만들고 디버깅을 하는데는 세 배 가량의 노력이 필요할 것으로 예상된다.

[예제 1-9] C 언어로 손수 만든 렉서

```
#include <stdio.h>
#include <ctype.h>
char *progname;
#define NUMBER 400
#define COMMENT 401
#define TEXT 402
#define COMMAND 403
main(argc,argv)
int argc;
char *argv[];
{
int val;
while(val = lexer()) printf("value is %d\n",val);
}
lexer()
{
   int c;

   while ((c=getchar()) == ' ' || c == '\t')
      ;
   if (c == EOF)
```

```
        return 0;
     if (c == '.' || isdigit(c)) {    /* 숫자 */
        while ((c = getchar()) != EOF && isdigit(c));
     if (c == '.') while ((c = getchar()) != EOF && isdigit(c));
        ungetc(c, stdin);
        return NUMBER;
     }
     if ( c == '#' ) { /* 주석 */
        int index = 1;
        while ((c = getchar()) != EOF && c != '\n');
        ungetc(c,stdin);
        return COMMENT;
     }
     if ( c == '"' ) { /* 문자 텍스트 */
        int index = 1;
        while ((c = getchar()) != EOF &&
       c != '"' && c != '\n');
        if(c == '\n') ungetc(c,stdin);
        return TEXT;
     }
     if ( isalpha(c)) { /* 명령어인지 여부를 검사한다. */
        int index = 1;
        while ((c = getchar()) != EOF && isalnum(c));
        ungetc(c, stdin);
        return COMMAND;
     }
     return c;
  }
```

[예제 1-10] 렉스로 만든 동일한 렉서

```
%{
#define NUMBER 400
#define COMMENT 401
#define TEXT 402
#define COMMAND 403
%}
%%
[ \t]+               ;
[0-9]+               |
[0-9]+ .[0-9]+       |
\.[0-9]+             { return NUMBER; }
#*                   { return COMMENT; }
\"[^\"\n]*\"         { return TEXT; }
[a-zA-Z][a-zA-Z0-9]+  { return COMMAND; }
\n                   { return '\n'; }
 return yytext[0];
%%
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
int val;
while(val = yylex()) printf("value is %d\n",val);
}
```

렉스는 손수 렉서를 작성할 경우 다루기 쉽지 않은 미묘한 내용도 자연스럽게 다루고 있다. 예를 들어, 여러분이 C 코드 안의 주석에 대한 처리를 건너뛰어야 한다고 가정하자. 주석의 맨 끝을 찾기 위해서 여러분은 "*"를 검색한 다음, 바로 뒤에 있는 문자가 "/"인지 여부를 검사할 것이다. 만약 "*" 다음에 "/" 문자가 위치한다면 주석의 끝을 만난 것이지만, 그렇지 않다면 계속해서 동일한 검색을 수행해야 한다. 이와 같은 C 렉서를 작성할 때 흔히 발생하는 실수 중의 하나는 "*"의 다음 문자 또한 "*" 이고, 이 두 번째 별표 뒤에 슬래시 기호가 오는 경우를 고려하지 않는 것이다. 이러한 실수는 결국 다음과 같이 쓰여진 주석을 제대로 인식하지 못한다는 사실을 의미한다(우리는 이 버그를 직접 작성하여 특정 야크 버전과 함께 배포한 렉서에서 발견하였다).

```
/** 주석 **/
```

여러분이 일단 렉스에 대해서 친숙해지면, 우리가 그랬던 것처럼, 렉스를 이용해서 렉서를 만드는 것이 훨씬 쉽기 때문에 결코 직접 렉서를 작성하려하지 않을 것이다.

2장에서는 렉스가 동작하는 방식에 대해서 좀더 자세하게 살펴볼 것이다. 3장에서는 야크에 대해 마찬가지로 자세히 살펴볼 것이다. 그 후, 렉스와 야크와 관련된 좀더 복잡한 문제점이나 특징들을 설명하기 위해서 몇 가지 커다란 예제들을 살펴보게 될 것이다.

## 연습 문제

1. 영어 파서를 확장하여 좀더 복잡한 문법 즉 전치사구, 형용사를 수식하는 부사 등도 다룰 수 있도록 만들어라.

2. 파서가 "has been"과 같은 복합 동사를 좀더 잘 다룰 수 있도록 만들어라. 새로운 단어나 혹은 보조 동사를 위한 토큰 타입인 AUXVERB 등을 추가해도 좋다.

3. "watch", "fly", "time" 혹은 "bear"와 같은 단어들은 언어의 문맥에 따라 다른 타입이 될 수도 있다. 이러한 상황을 어떻게 다룰 것인가? 새로운 단어와 토큰 타입 NOUN_OR_VERB를 추가하여, **subject**, **verb** 그리고 **object** 규칙의 대안으로 만들어라. 이러한 새로운 규칙은 제대로 동작하는가?

4. 사람들은 낯선 단어를 듣는 경우에, 대개 그 단어가 전체적인 문맥 속에서 언어의 어떤 요소로 사용되는지 추측함으로써 의미를 파악하려고 한다. 렉서는 새로운 단어의 타입을 동적으로 파악할 수 있는가? 예를 들면 "ing"으로 끝나는 단어는 대개 동사일 가능성이 높으며, 앞에 "a" 혹은 "the"가 놓인 단어는 명사이거나 형용사일 가능성이 많다.

5. 렉스와 야크는 진짜 영어 문법을 만들기 위해서도 좋은 도구일까? 아니라면 그 이유는?