```
import java.io.*;
class Parser {
    static int lookahead;

    public Parser() throws IOException {
        lookahead = System.in.read();
    }

    void expr() throws IOException {
        term();
        while(true) {
            if( lookahead == '+' ) {
                match('+'); term(); System.out.write('+');
            }
            else if( lookahead == '-' ) {
                match('-'); term(); System.out.write('-');
            }
            else return;
        }
    }

    void term() throws IOException {
        if( Character.isDigit((char)lookahead) ) {
            System.out.write((char)lookahead); match(lookahead);
        }
        else throw new Error("syntax error");
    }

    void match(int t) throws IOException {
        if( lookahead == t ) lookahead = System.in.read();
        else throw new Error("syntax error");
    }
}

public class Postfix {
    public static void main(String[] args) throws IOException {
        Parser parse = new Parser();
        parse.expr(); System.out.write('\n');
    }
}
```

Figure 2.27: Java program to translate infix expressions into postfix form

A Few Salient Features of Java

Those unfamiliar with Java may find the following notes on Java helpful in reading the code in Fig. 2.27:

- A class in Java consists of a sequence of variable and function definitions.
- Parentheses enclosing function parameter lists are needed even if there are no parameters; hence we write `expr()` and `term()`. These functions are actually procedures, because they do not return values, signified by the keyword `void` before the function name.
- Functions communicate either by passing parameters "by value" or by accessing shared data. For example, the functions `expr()` and `term()` examine the lookahead symbol using the class variable `lookahead` that they can all access since they all belong to the same class `Parser`.
- Like C, Java uses `=` for assignment, `==` for equality, and `!=` for inequality.
- The clause "`throws IOException`" in the definition of `term()` declares that an exception called `IOException` can occur. Such an exception occurs if there is no input to be read when the function `match` uses the routine `read`. Any function that calls `match` must also declare that an `IOException` can occur during its own execution.

2.6 Lexical Analysis

A lexical analyzer reads characters from the input and groups them into "token objects." Along with a terminal symbol that is used for parsing decisions, a token object carries additional information in the form of attribute values. So far, there has been no need to distinguish between the terms "token" and "terminal," since the parser ignores the attribute values that are carried by a token. In this section, a token is a terminal along with additional information.

A sequence of input characters that comprises a single token is called a *lexeme*. Thus, we can say that the lexical analyzer insulates a parser from the lexeme representation of tokens.

The lexical analyzer in this section allows numbers, identifiers, and "white space" (blanks, tabs, and newlines) to appear within expressions. It can be used to extend the expression translator of the previous section. Since the expression grammar of Fig. 2.21 must be extended to allow numbers and identifiers, we

shall take this opportunity to allow multiplication and division as well. The extended translation scheme appears in Fig. 2.28.

$$
\begin{array}{lcll}
\mathit{expr} & \rightarrow & \mathit{expr} + \mathit{term} & \{\ \mathrm{print}('+')\ \} \\
 & | & \mathit{expr} - \mathit{term} & \{\ \mathrm{print}('-')\ \} \\
 & | & \mathit{term} & \\
 & & & \\
\mathit{term} & \rightarrow & \mathit{term} * \mathit{factor} & \{\ \mathrm{print}('*')\ \} \\
 & | & \mathit{term} / \mathit{factor} & \{\ \mathrm{print}('/')\ \} \\
 & | & \mathit{factor} & \\
 & & & \\
\mathit{factor} & \rightarrow & (\ \mathit{expr}\ ) & \\
 & | & \mathbf{num} & \{\ \mathrm{print}(\mathbf{num}.\mathit{value})\ \} \\
 & | & \mathbf{id} & \{\ \mathrm{print}(\mathbf{id}.\mathit{lexeme})\ \}
\end{array}
$$

Figure 2.28: Actions for translating into postfix notation

In Fig. 2.28, the terminal **num** is assumed to have an attribute **num**.*value*, which gives the integer value corresponding to this occurrence of **num**. Terminal **id** has a string-valued attribute written as **id**.*lexeme*; we assume this string is the actual lexeme comprising this instance of the token **id**.

The pseudocode fragments used to illustrate the workings of a lexical analyzer will be assembled into Java code at the end of this section. The approach in this section is suitable for hand-written lexical analyzers. Section 3.5 describes a tool called Lex that generates a lexical analyzer from a specification. Symbol tables or data structures for holding information about identifiers are considered in Section 2.7.

2.6.1 Removal of White Space and Comments

The expression translator in Section 2.5 sees every character in the input, so extraneous characters, such as blanks, will cause it to fail. Most languages allow arbitrary amounts of white space to appear between tokens. Comments are likewise ignored during parsing, so they may also be treated as white space.

If white space is eliminated by the lexical analyzer, the parser will never have to consider it. The alternative of modifying the grammar to incorporate white space into the syntax is not nearly as easy to implement.

The pseudocode in Fig. 2.29 skips white space by reading input characters as long as it sees a blank, a tab, or a newline. Variable *peek* holds the next input character. Line numbers and context are useful within error messages to help pinpoint errors; the code uses variable *line* to count newline characters in the input.

```
for ( ; ; peek = next input character ) {
        if ( peek is a blank or a tab ) do nothing;
        else if ( peek is a newline ) line = line+1;
        else break;
}
```

Figure 2.29: Skipping white space

2.6.2 Reading Ahead

A lexical analyzer may need to read ahead some characters before it can decide on the token to be returned to the parser. For example, a lexical analyzer for C or Java must read ahead after it sees the character `>`. If the next character is `=`, then `>` is part of the character sequence `>=`, the lexeme for the token for the "greater than or equal to" operator. Otherwise `>` itself forms the "greater than" operator, and the lexical analyzer has read one character too many.

A general approach to reading ahead on the input, is to maintain an input buffer from which the lexical analyzer can read and push back characters. Input buffers can be justified on efficiency grounds alone, since fetching a block of characters is usually more efficient than fetching one character at a time. A pointer keeps track of the portion of the input that has been analyzed; pushing back a character is implemented by moving back the pointer. Techniques for input buffering are discussed in Section 3.2.

One-character read-ahead usually suffices, so a simple solution is to use a variable, say *peek*, to hold the next input character. The lexical analyzer in this section reads ahead one character while it collects digits for numbers or characters for identifiers; e.g., it reads past `1` to distinguish between `1` and `10`, and it reads past `t` to distinguish between `t` and `true`.

The lexical analyzer reads ahead only when it must. An operator like `*` can be identified without reading ahead. In such cases, *peek* is set to a blank, which will be skipped when the lexical analyzer is called to find the next token. The invariant assertion in this section is that when the lexical analyzer returns a token, variable *peek* either holds the character beyond the lexeme for the current token, or it holds a blank.

2.6.3 Constants

Anytime a single digit appears in a grammar for expressions, it seems reasonable to allow an arbitrary integer constant in its place. Integer constants can be allowed either by creating a terminal symbol, say **num**, for such constants or by incorporating the syntax of integer constants into the grammar. The job of collecting characters into integers and computing their collective numerical value is generally given to a lexical analyzer, so numbers can be treated as single units during parsing and translation.