Multi Character Lookahead

We may allow finite automata to look beyond the next input character.

This feature is necessary to implement a scanner for FORTRAN.

In FORTRAN, the statement

```
DO 10 J = 1,100
```

specifies a loop, with index **J** ranging from 1 to 100.
The statement

```
DO 10 J = 1.100
```

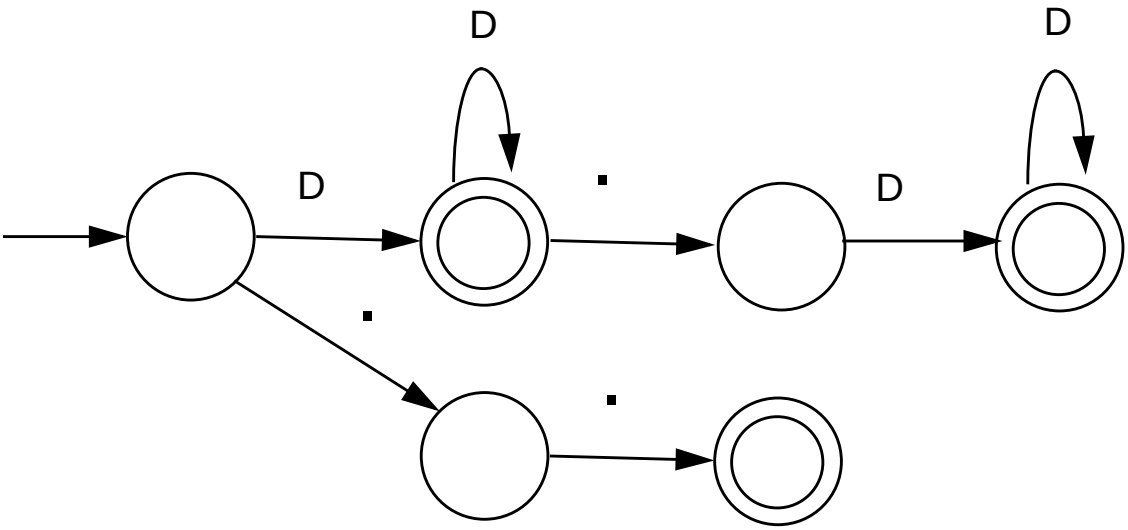
is an assignment to the variable **DO10J**. (Blanks are not significant except in strings.)

A FORTRAN scanner decides whether the **O** is the last character of a **DO** token only after reading as far as the comma (or period).

A milder form of extended lookahead problem occurs in Pascal and Ada.
The token **`10.50`** is a real literal, whereas **`10..50`** is three different tokens.

We need two-character lookahead after the **`10`** prefix to decide whether we are to return **`10`** (an integer literal) or **`10.50`** (a real literal).

Suppose we use the following FA.



Given **`10..100`** we scan three characters and stop in a non-accepting state.

Whenever we stop reading in a non-accepting state, we *back up* along accepted characters until an accepting state is found.

Characters we back up over are *rescanned* to form later tokens. If no accepting state is reached during backup, we have a lexical error.

Performance Considerations

Because scanners do so much character-level processing, they can be a real performance bottleneck in production compilers.

Speed is not a concern in our project, but let's see why scanning speed can be a concern in production compilers.

Let's assume we want to compile at a rate of 1000 lines/sec. (so that most programs compile in just a few seconds).

Assuming 30 characters/line (on average), we need to scan 30,000 char/sec.

On a 30 SPECmark machine (30 million instructions/sec.), we have 1000 instructions per character to spend on *all* compiling steps.

If we allow 25% of compiling to be scanning (a compiler has a lot more to do than just scan!), that's just 250 instructions per character.

A key to efficient scanning is to group character-level operations whenever possible. It is better to do one operation on n characters rather than n operations on single characters.

In our examples we've read input one character as a time. A subroutine call can cost hundreds or thousands of instructions to execute—far too much to spend on a single character.

We prefer routines that do block reads, putting an entire block of characters directly into a buffer.

Specialized scanner generators can produce particularly fast scanners.

The GLA scanner generator claims that the scanners it produces run as fast as:

```
while(c != Eof) {
   c = getchar();
}
```

Lexical Error Recovery

A character sequence that can't be scanned into any valid token is a *lexical error*.

Lexical errors are uncommon, but they still must be handled by a scanner. We won't stop compilation because of so minor an error.

Approaches to lexical error handling include:

- Delete the characters read so far and restart scanning at the next unread character.
- Delete the first character read by the scanner and resume scanning at the character following it.

Both of these approaches are reasonable.

The first is easy to do. We just reset the scanner and begin scanning anew.

The second is a bit harder but also is a bit safer (less is immediately deleted). It can be implemented using scanner backup.

Usually, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

(Why at the beginning?)

In these case, the two approaches are equivalent.

The effects of lexical error recovery might well create a later *syntax error*, handled by the parser.

Consider

```
...for$tnight...
```

The **$** terminates scanning of **for**. Since no valid token begins with **$**, it is deleted. Then **tnight** is scanned as an identifier. In effect we get

```
...for tnight...
```

which will cause a syntax error. Such “false errors” are unavoidable, though a syntactic error-repair may help.

Error Tokens

Certain lexical errors require special care. In particular, runaway strings and runaway comments ought to receive special error messages.

In Java strings may not cross line boundaries, so a runaway string is detected when an end of a line is read within the string body. Ordinary recovery rules are inappropriate for this error. In particular, deleting the first character (the double quote character) and restarting scanning is a *bad* decision.

It will almost certainly lead to a cascade of "false" errors as the string text is inappropriately scanned as ordinary input.

One way to handle runaway strings is to define an *error token*.

An error token is *not* a valid token; it is never returned to the parser. Rather, it is a *pattern* for an error condition that needs special handling. We can define an error token that represents a string terminated by an end of line rather than a double quote character.

For a valid string, in which internal double quotes and back slashes are escaped (and no other escaped characters are allowed), we can use

" (Not(" | Eol | \ ) | \" | \\ )* "

For a runaway string we use

" (Not(" | Eol | \ ) | \" | \\ )* Eol

(**Eol** is the end of line character.)

When a runaway string token is recognized, a special error message should be issued.

Further, the string may be "repaired" into a correct string by returning an ordinary string token with the closing Eol replaced by a double quote.

This repair may or may not be "correct." If the closing double quote is truly missing, the repair will be good; if it is present on a succeeding line, a cascade of inappropriate lexical and syntactic errors will follow.

Still, we have told the programmer exactly what is wrong, and that is our primary goal.

In languages like C, C++, Java and CSX, which allow multiline comments, improperly terminated (runaway) comments present a similar problem.

A runaway comment is not detected until the scanner finds a close comment symbol (possibly belonging to some other comment) or until the end of file is reached. Clearly a special, detailed error message is required.

Let's look at Pascal-style comments that begin with a `{` and end with a `}`. Comments that begin and end with a pair of characters, like `/*` and `*/` in Java, C and C++, are a bit trickier.

Correct Pascal comments are defined quite simply:

{ Not(})* }

To handle comments terminated by **Eof**, this error token can be used:

{ Not(})* Eof

We want to handle comments unexpectedly closed by a close comment belonging to another comment:

```
{... missing close comment
 ... { normal comment }...
```

We will issue a *warning* (this form of comment is lexically legal).

Any comment containing an open comment symbol in its body is most probably a missing **}** error.

We split our legal comment definition into two token definitions.

The definition that accepts an open comment in its body causes a warning message ("Possible unclosed comment") to be printed.

We now use:

{ Not({ | })* } and
{ (Not({ | })* { Not({ | })*)+ }

The first definition matches correct comments that do not contain an open comment in their body.

The second definition matches correct, but suspect, comments that contain at least one open comment in their body.

Single line comments, found in Java, CSX and C++, are terminated by Eol.

They can fall prey to a more subtle error—what if the last line has no Eol at its end?

The solution?

Another error token for single line comments:

// Not(Eol)*

This rule will only be used for comments that don't end with an Eol, since scanners always match the longest rule possible.