Identifiers vs. Reserved Words

Most programming languages contain *reserved words* like **`if`**, **`while`**, **`switch`**, etc. These tokens look like ordinary identifiers, but aren't.

It is up to the scanner to decide if what looks like an identifier is really a reserved word. This distinction is vital as reserved words have different token codes than identifiers and are parsed differently.

How can a scanner decide which tokens are identifiers and which are reserved words?

- We can scan identifiers and reserved words using the same pattern, and then look up the token in a special "reserved word" table.

- It is known that any regular expression may be *complemented* to obtain all strings not in the original regular expression. Thus $\overline{\mathbf{A}}$, the complement of **A**, is regular if **A** is. Using complementation we can write a regular expression for nonreserved identifiers: $(\overline{\text{ident}}|\text{if}|\text{while}|\ldots)$ Since scanner generators don't usually support complementation of regular expressions, this approach is more of theoretical than practical interest.
- We can give distinct regular expression definitions for each reserved word, and for identifiers. Since the definitions overlap (**`if`** will match a reserved word *and* the general identifier pattern), we give

priority to reserved words. Thus a token is scanned as an identifier if it matches the identifier pattern *and* does not match any reserved word pattern. This approach is commonly used in scanner generators like Lex and JLex.

Converting Token Values

For some tokens, we may need to convert from string form into numeric or binary form.

For example, for integers, we need to transform a string a digits into the internal (binary) form of integers.

We know the format of the token is valid (the scanner checked this), but:

- The string may represent an integer too large to represent in 32 or 64 bit form.
- Languages like CSX and ML use a non-standard representation for negative values (**`~123`** instead of **`-123`**)

We can safely convert from string to integer form by first converting the string to double form, checking against max and min int, and then converting to int form if the value is representable.

Thus **d** = **new Double(str)** will create an object **d** containing the value of **str** in double form. If **str** is too large or too small to be represented as a double, plus or minus infinity is automatically substituted.

d.doubleValue() will give **d**'s value as a Java double, which can be compared against **Integer.MAX_VALUE** or **Integer.MIN_VALUE**.

If **d.doubleValue()** represents a valid integer,
(int) d.doubleValue()
will create the appropriate integer value.

If a string representation of an integer begins with a "~" we can strip the "~", convert to a double and then negate the resulting value.

Scanner Termination

A scanner reads input characters and partitions them into tokens.

What happens when the end of the input file is reached? It may be useful to create an **Eof** pseudo-character when this occurs. In Java, for example, **InputStream.read()**, which reads a single byte, returns -1 when end of file is reached. A constant, **EOF**, defined as -1 can be treated as an "extended" ASCII character. This character then allows the definition of an **Eof** token that can be passed back to the parser.

An **Eof** token is useful because it allows the parser to verify that the logical end of a program corresponds

to its physical end. Most parsers *require* an end of file token.

Lex and Jlex automatically create an **Eof** token when the scanner they build tries to scan an **EOF** character (or tries to scan when **eof()** is true).

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
```

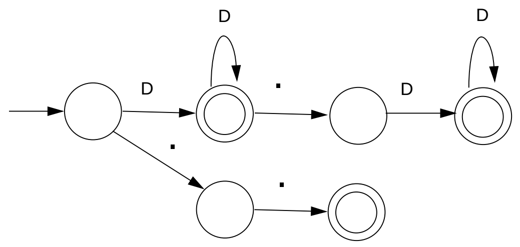
is an assignment to the variable **DO10J**. (Blanks are not significant except in strings.)

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.