Lexical errors

If no prefix of the input string forms a valid token, a lexical error has occurred. When this happens, the lexical will usually report an error. At this point, it may stop reading the input or it may attempt continued lexical analysis by skipping characters until a valid prefix is found. The purpose of the latter approach is to try finding further lexical errors in the same input, so several of these can be corrected by the user before re-running the **Lexical**. Some of these subsequent errors may, however, not be real errors but may be caused by the lexical not skipping enough characters (or skipping too many) after the first error is found. If, for example, the start of a comment is ill-formed, the lexical may try to interpret the contents of the comment as individual tokens, and if the end of a comment is ill-formed, the lexical will read until the end of the next comment (if any) before continuing, hence skipping too much text.

When the lexical finds an error, the consumer of the tokens that the lexical produces (e.g., the rest of the compiler) cannot usually itself produce a valid result. However, the compiler may try to find other errors in the remaining input, again allowing the user to find several errors in one edit-compile cycle. Again, some of the subsequent errors may really be spurious errors caused by lexical error(s), so the user will have to guess at the validity of every error message except the first, as only the first error message is guaranteed to be a real error. Nevertheless, such error recovery has, when the input is so large that restarting the lexical from the start of input incurs a considerable time overhead, proven to be an aid in productivity by locating more errors in less time. Less commonly, the lexical may work interactively with a text editor and restart from the point at which an error was spotted after the user has tried to _x the error.

General view about errors

One of the important tasks that a compiler must perform is the detection of and recovery from errors. Recovery from errors is important, because the compiler will be scanning and compiling the entire program, perhaps in the presence of errors; so as many errors as possible need to be detected.

Every phase of a compilation expects the input to be in a particular format, and whenever that input is not in the required format, an error is returned. When detecting an error, a compiler scans some of the tokens that are ahead of the error's point of occurrence. The fewer the number of tokens that must be scanned ahead of the point of error occurrence, the better the compiler's error-detection capability. For example, consider the following statement:

if a = b then x: = y +z;

The error in the above statement will be detected in the syntactic analysis phase, but not before the syntax analyzer sees the token "then"; but the first token, itself, is in error. After detecting an error, the first thing that a compiler is supposed to do is to report the error by producing a suitable diagnostic. A good error diagnostic should possess the following properties.

1. The message should be produced in terms of the original source program rather than in terms of some internal representation of the source program. For example, the message should be produced along with the line numbers of the source program.
2. The error message should be easy to understand by the user.

3. The error message should be specific and should localize the problem. For example, an error message should read, "x is not declared in function fun," and not just, "missing declaration".

4. The message should not be redundant; that is, the same message should not be produced again and again.

Therefore, a compiler should report errors by generating messages with the above properties. The errors captured by the compiler can be classified as either syntactic errors or semantic errors. Syntactic errors are those errors that are detected in the lexical or syntactic analysis phase by the compiler. Semantic errors are those errors detected by the compiler.

Recovery From Lexical Phase Errors

The lexical analyzer detects an error when it discovers that an input's prefix

does not fit the specification of any token class. After detecting an error, the lexical analyzer can invoke an error recovery routine. This can entail a variety of remedial actions.

The simplest possible error recovery is to skip the erroneous characters until the lexical analyzer finds another token. But this is likely to cause the parser to read a deletion error, which can cause severe difficulties in the syntax analysis and remaining phases. One way the parser can help the lexical analyzer can improve its ability to recover from errors is to make its list of legitimate tokens (in the current context) available to the error recovery routine. The error-recovery routine can then decide whether a remaining input's prefix matches one of these tokens closely enough to be treated as that token.

lexical **generators**

A lexical generator will typically use a notation for regular expressions similar to the one described earlier , but may require alphabet characters to be quoted to distinguish them from the symbols used to build regular expressions.

The input to the lexical generator will normally contain a list of regular expressions that each denote a token. Each of these regular expressions has an associated action. The action describes what is passed on to the consumer (e.g., the parser), typically an element from a token data type, which describes the type of token (NUM, ID, etc.) and sometimes additional information such as the value of a number token, the name of an identifier token and, perhaps, the position of the token in the input file. The information needed to construct such values is typically provided by

Normally, the lexical generator requires white-space and comments to be defined by regular expressions. The actions for these regular expressions are typically empty, meaning that white-space and comments are just ignored.

An action can be more than just returning a token. If, for example, a language has a large number of keywords, then a DFA that recognizes all of these individually can be fairly large. In such cases, the keywords are not described as separate regular expressions in the lexical definition but instead treated as special cases of the identifier token. The action for identifiers will then look the name up in a table of keywords and return the appropriate token type (or an identifier token if the name is not a keyword).

A similar strategy can be used if the language allows identifiers to shadow keywords. Another use of non-trivial lexical actions is for nested comments. In principle, a regular expression (or finite automaton) cannot recognize arbitrarily nested comments, but by using a global counter, the actions for comment tokens can keep track of the nesting level. If escape sequences (for defining, e.g., control characters) are allowed in string

constants, the actions for string tokens will, typically, translate the string containing these sequences into a string where they have been substituted by the characters they represent.