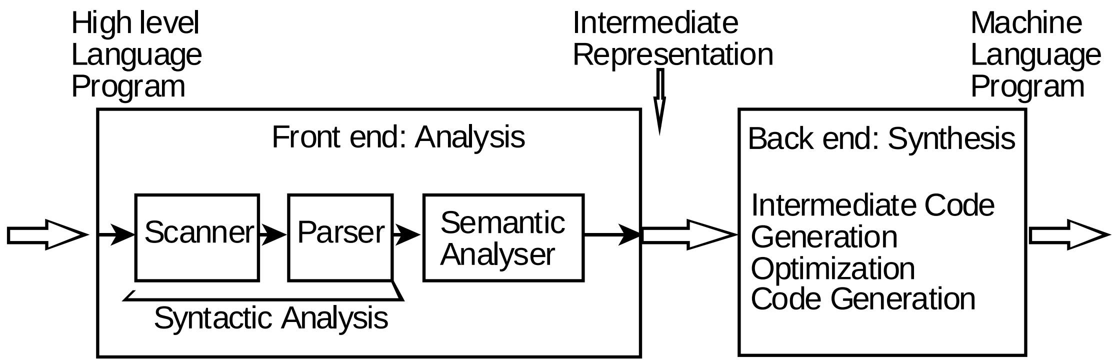
Lexical Analysis

Here is an overview of a compiler:



And a more detailed view:

Source Code ➔

Pre-processor

➔ *Modified source Code* ➔

Analysis
(source code dependent)

Scanner

➔ *Token stream* ➔

Parser

➔ *Syntax Tree* ➔

Semantic Analyzer

➔ *Annoted Syntax Tree* ➔

Intermediate Code Generator

➔ *Intermediate Representation* ➔

Synthesis
(target code dependent)

Machine-Independent Code Optimizer

➔ *Intermediate Representation* ➔

Code Generator

➔ *Target-machine Code* ➔

Machine-Dependent Code Optimizer

➔ *Target-machine Code*

The various steps in the compiling process are called phases.

Token: the smallest element of a programming language. It is made up of characters but it is "atomic" meaning that it can't be broken down into characters. Tokens include at least the following:

- reserved words,
- punctuation marks,
- identifiers,
- literals.

Lexical Analysis/Scanning: the first and simplest phase of a compiler. It assembles the characters into tokens.

For example, given the following piece of Java source code:

```
if   (i==12)  age> 1
```

the lexical analyzer will produce the following token sequence:

Keyword IF	Operator LPAREN	Identifier i	Operator EQUAL	Number 12	Operator RPAREN	Identifier age	Operator GT	Number 1

Tokens are encoded in a way which makes subsequent processing easy using a fixed format, usually a number, or pair of numbers, or equivalent:

- for fixed element of the language (e.g. keywords, operators) a single number may be used. For example the token "if" might be denoted by 22.while the token "{" might be denoted by 16.
- for user defined categories (e.g. identifiers, literals) a pair may be used that identifies the category accompanied by another item identifying the specific element in the category. For example, the identifier xyz might be denoted by 57 (used for all identifiers) and 45 (specific to xyz, usually indicating where the specific identifier information is to be found, usually in a symbol table).

Since the input is a stream of characters it is sometimes difficult to determine where one token finishes and another starts. Clearly one could require that the tokens be separated by spaces. Most programming languages allow this but, as you know, do not require it. For example, the code:

foo=55

is an acceptable assignment and a scanner will be able to distinguish the three tokens “foo”, “=”, and “55”. However, there are situations where it can be trickier. For example:

ifp=5

Should the scanner consider “ifp” to be a single token (i.e. an identifier), or should it look at it as the keyword “if”, followed by the identifier “p”? This made the specification of some earlier programming languages such as FORTRAN and COBOL more difficult. In modern programming languages the ***principle of the longest string*** usually applies. It states that, if there are two possible interpretations, the longest one will be chosen. In the above example the scanner will choose the “ifp” interpretation, since it is longer than “if”. In a case where the intent is to have two tokens “if” and “g” they must be separated by one or more spaces.

Hence the scanner will accumulate characters one at a time and add it to the current token until one of the following happens:

- a white space is encountered (i.e. space, <CR>, <tab>, <newline>).
- the token would become illegal.

At this point the scanner terminates the current token and start on a new one.

Exercise:
Imagine you have a language that has only operators, identifiers, and two reserved words, and the following characteristics:

- the only legal characters are the digits 0 and 1, the letter K, and the characters +
- there are only two operators + and ++
- there are only two keywords K0 and K1 (keywords have precedence over identifiers)
- identifiers can be of any length, and can only include K’s, 0's, and 1's.
- the principle of the longest string applies.
- spaces are legal, but have no other significance.

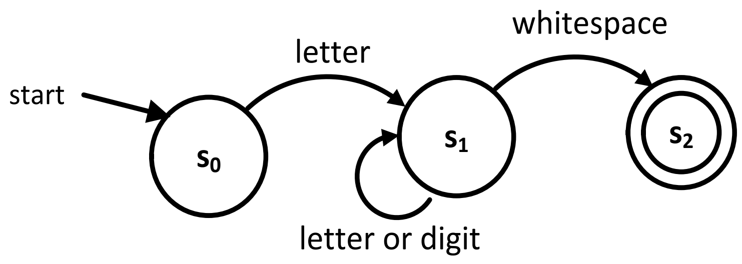
Show the final sequence of tokens resulting from a lexical scan of the following character sequence

001K111+++K0 1K1K0K1

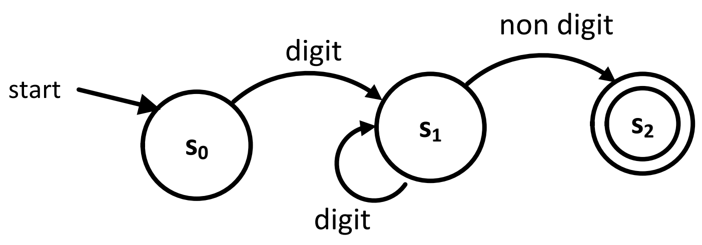
Tokens are usually specified using regular expressions and can be identified using finite automata (or equivalent).

Examples: Here are some automata which represent the analysis of various tokens.

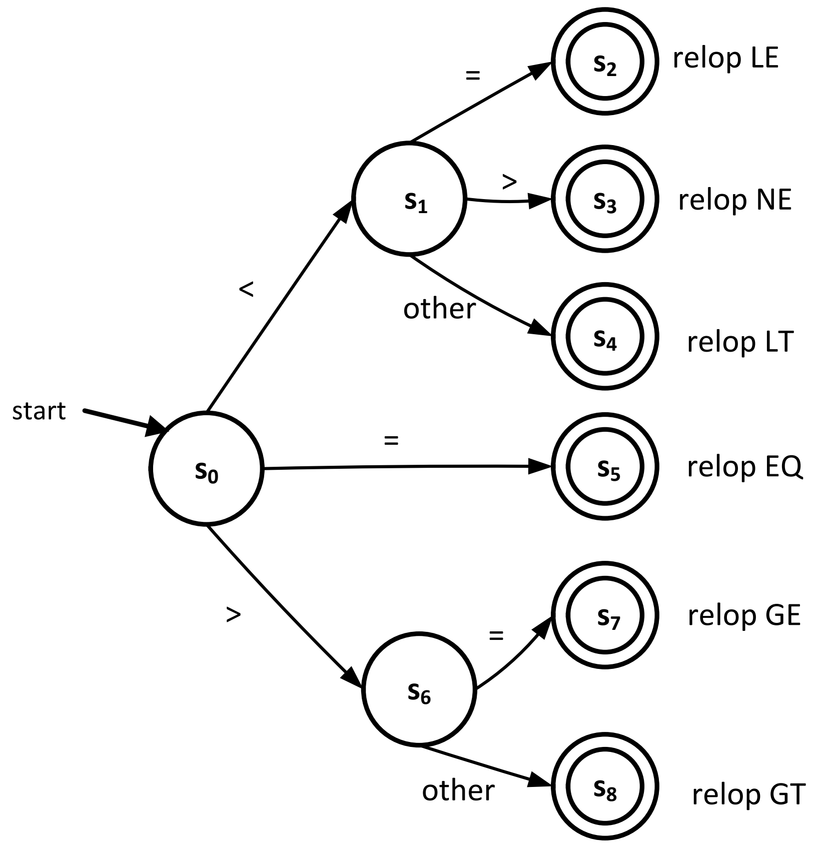
Identifier



unsigned integer



relational operators



Note that to represent an actual scanner all those automata would have to be combined. Notice that in some cases the character should not be consumed and should be put back in the stream of characters (i.e. “other” in the relop case). This generally requires the use of a “buffer” which can be emptied if the character is consumed but that will hold the character otherwise for use on the next token. .

Scanners can be written by hand modeling a finite state automaton or one can use scanner generators such as Lex, FLex, or JLex. All those scanners rely on regular expressions to describe the syntax of the tokens