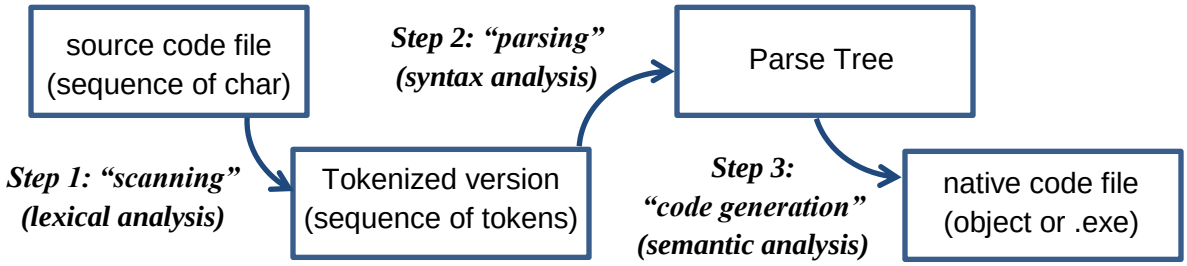
3.0 Lexical Analysis

First, let's see a simplified overview of the compilation process:



"Scanning" == converting the programmers original source code file, which is typically a sequence of *ASCII characters*, into a sequence of *tokens*.

"Token" = a single atomic element of the programming language.

Consider the following piece of *Java* source code:

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

This code fragment contains 19 characters. It would be very difficult to formally specify the syntactic structure of a language such as Java entirely based on the ASCII characters. Instead, the syntax of Java is described based on "tokens" in the language, and a separate lexical definition exists that defines how the characters in the source code translate into tokens.

In the above case, scanning would produce a sequence of tokens such as:

Keyword IF	Operator LParen	Identifier `I`	Operator EQLTY	Number `12`	Operator RParen	Identifier `match`	Operator GTE	Number `1`

The sequence of tokens may be represented as a list, where each token includes (1) the type, and (2) the actual value or text ("lexeme"), as shown. Although lexical errors can occur, it is common for a program to be lexically correct, but still contain syntax errors that will be detected during parsing. We will examine the syntax structure and parsing of programs later. But first, let's look at the *lexical* structure, and how *scanning* is done.

Scanning is the simplest of the compilation steps above. It can generally be done in 1 or 2 passes through the source code. The trick is figuring out when one token ends and the next one begins.

One way of separating tokens would be to require the programmer to put spaces between them. Most languages allow this, but don't require it in every case. For example, the code:

```
foo=35;
```

is in almost every language an acceptable way of indicating an assignment statement. The scanner is capable of knowing that the "**foo**" and the "**=**" are for different tokens, even though there is no space separating them.

However, consider the following trickier case:

```
ifg=3;
```

Should the lexical analyzer interpret this as starting with a single token – a variable named "`ifg`"? Or should it interpret it as starting with the keyword "`if`" followed by the variable "`g`"? This dilemma complicated the lexical specifications of early programming languages such as Fortran and Cobol.

Principle of Longest Substring: The above dilemma is solved through the use of an elegant technique. Briefly stated, given a choice between two interpretations, we always choose the *longest* one. So in the above example, the correct interpretation is "`ifg`", since that is longer than "`if`". If the programmer had actually *intended* "`if`" and "`g`" to be separate, it is his/her responsibility to include a space to separate the "`if`" from the "`g`".

In summary, the scanner collects characters one-by-one, adding them to the current token, until one of the following occurs:

- whitespace (i.e., space, <cr>, <tab>, <newline>) is encountered, or
- the token becomes illegal.

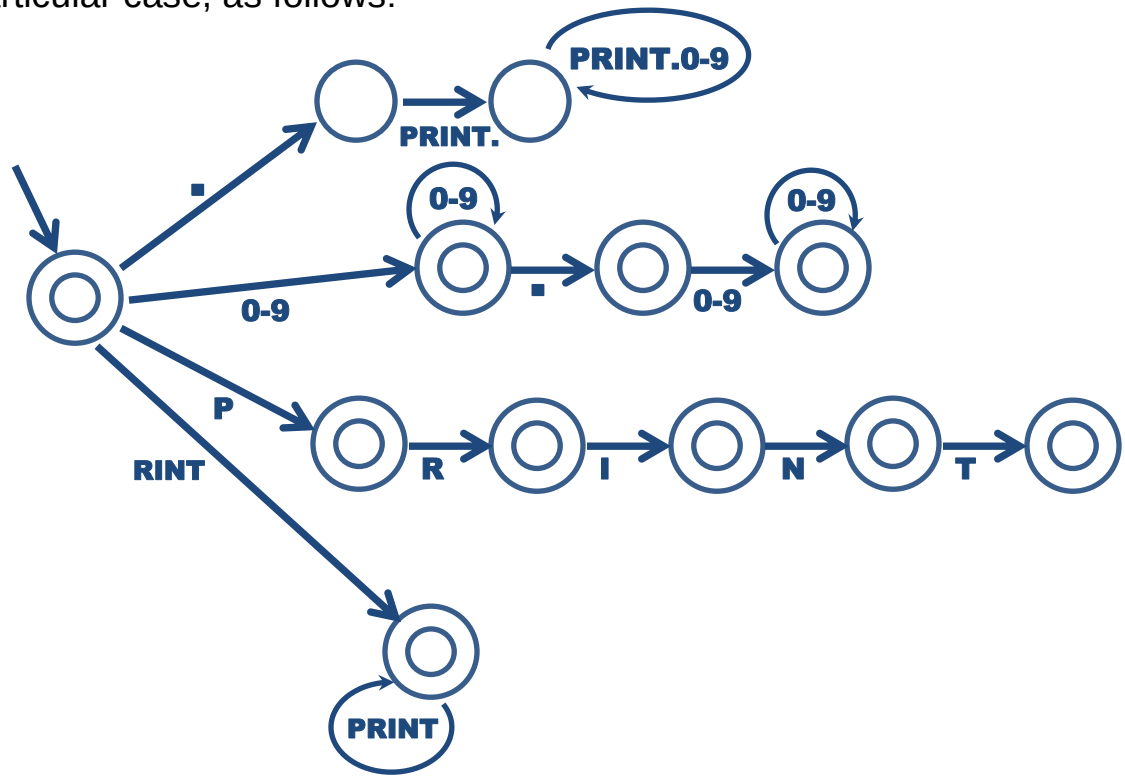
At that time, the scanner ends the current token and starts building another. The process continues until the entire source file is converted to tokens.

Formal lexical specification can be done with a grammar. Although, it is often simple enough that it can be done with a finite automaton.

Example:
Consider the following *lexical* requirements for a very simple language:

- the alphabet includes the digits `0-9`, characters `P`, `R`, `I`, `N`, `T`, and the period (.)
- there is one keyword: "`PRINT`"
- identifiers can be any other sequence of the `P`, `R`, `I`, `N`, `T` characters
- numbers can be any sequence of digits, with an optional decimal point. If there is a decimal point, there must be at least one digit before or after the decimal.
- The *principle of longest substring* applies.

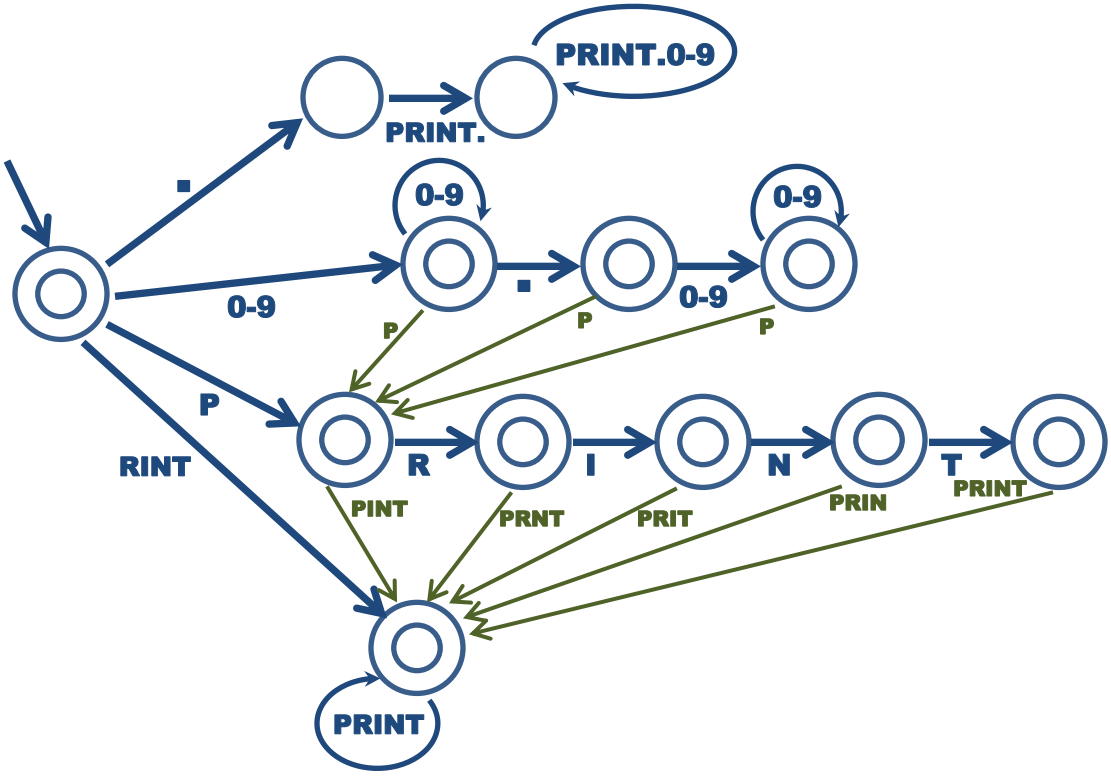
Suppose we wish to express this specification formally with an automaton. Using the skeleton method, we would start by creating a skeleton for each particular case, as follows:



The skeleton shown above handles, from top to bottom: (1) the illegal use of the decimal point without any adjacent digits, (2) legal numbers, (3) the keyword PRINT, and (4) identifiers. Of course, as shown it doesn't handle *sequences of tokens*. For that, we need to add more arcs, as follows.

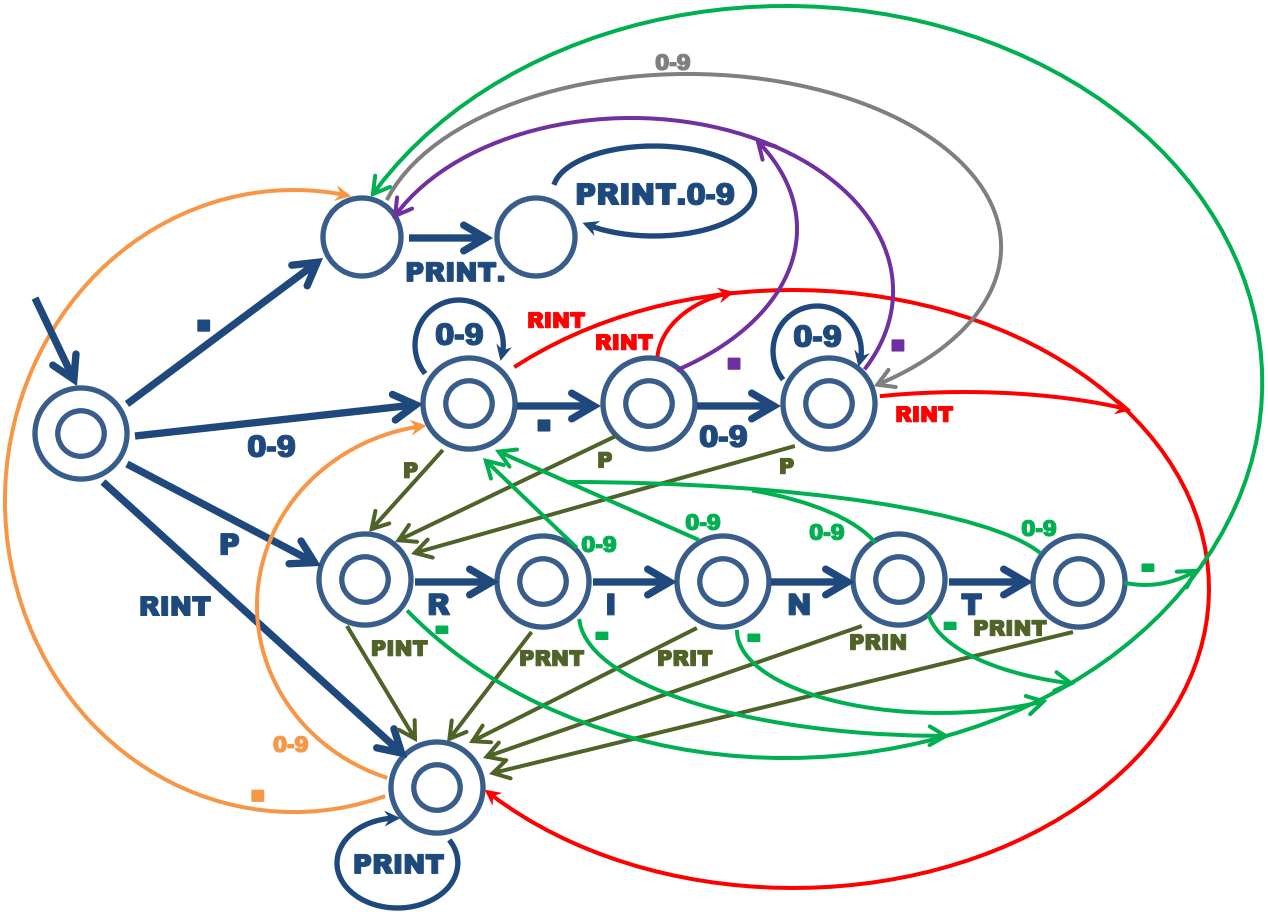
Consider the case where a valid number is followed by a keyword. Since a keyword always starts with a "`P`", arcs would need to be added from the states for numbers, to the states that check for the keyword `PRINT`.

Similarly, consider an identifier such as "`PRIPP`". It starts out with some of the same first few letters as the keyword `PRINT`. Arcs need to be added to allow for that possibility. The necessary added arcs for both of the above cases are shown in dark green as follows:



And there are several other cases. The next version of the diagram shows the same skeleton, with arcs handling several other cases, including:

- a valid number followed by an identifier (shown in red)
- a keyword followed by a number (shown in light green)
- a number followed immediately by another number (shown in purple)
- a number containing only digits to the right of the decimal, without any digits before the decimal (shown in grey)
- an identifier followed by a number (shown in orange)



The automaton is complete when *every state has an outgoing arc for every symbol in the alphabet*. In the above figure, **every** state has outgoing arcs to handle **all** of the symbols `P`, `R`, `I`, `N`, `T`, `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, and the period (`.`). You don't need to memorize diagrams like the one shown above; the important points for you to understand are:

1. the process used to create the finite automaton,
2. that an automaton is often sufficiently powerful for the lexical phase, and
3. for the automaton to be complete, **every** state must have an outgoing arc for **each** symbol in the language's alphabet.

Finally, note that there is one detail still missing from the above diagram: whitespace! How would that be handled? *[hint: draw an arc from every accepting state back to the start state, and label those arcs with whitespace].*