Finite Automata and Regular Expressions

- The ***scanner*** (also sometimes called the ***lexical analyzer***, or simply the ***lexer***) is the first phase of a compiler's front end.

- The scanner is responsible for reading the stream of characters comprising the source code and transforming it into a stream of words in the source language, where each word is classified with a syntactic category, or "part of speech" (e.g. `identifier`, `number`, `if`, `{`, `=`, etc.).

- There are a few different sub-tasks involved in scanning:
    - Aggregate characters into words.
    - Determine whether each word is legal in the source language.
    - If the word is valid, assign it a syntactic category.

- To accomplish these tasks, the scanner applies a set of rules known as the ***microsyntax***, which specify the lexical structure of the source language.

- The microsyntax for a language describes, e.g.:
    - How to group characters into words in the language.
    - How to separate words that run together.
        - E.g. `a=b+c-1;` is a valid C++ statement containing 8 distinct words, none of which are separated by whitespace.
    - How to apply syntactic categories to words.

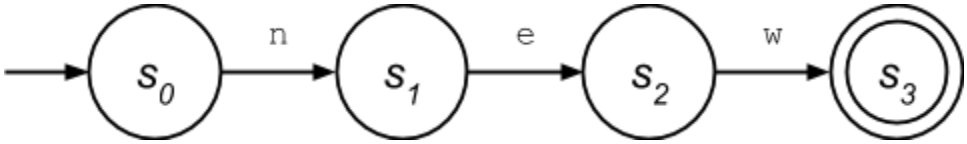
- To accomplish these tasks, a scanner typically makes use of ***finite automata*** and ***regular expressions***. We will explore these topics here.

Recognizing words with finite automata

- Consider the problem of recognizing only the keyword `new`. Assuming we have available a function `nextChar()` that returns the next character in the input source code stream, we might write code like this to recognize the `new` keyword:

```
c = nextChar()
if (c == 'n')
  c = nextChar()
  if (c == 'e')
    c = nextChar()
    if (c == 'w')
      return SUCCESS
    else
      return FAILURE
  else
    return FAILURE
else
  return FAILURE
```

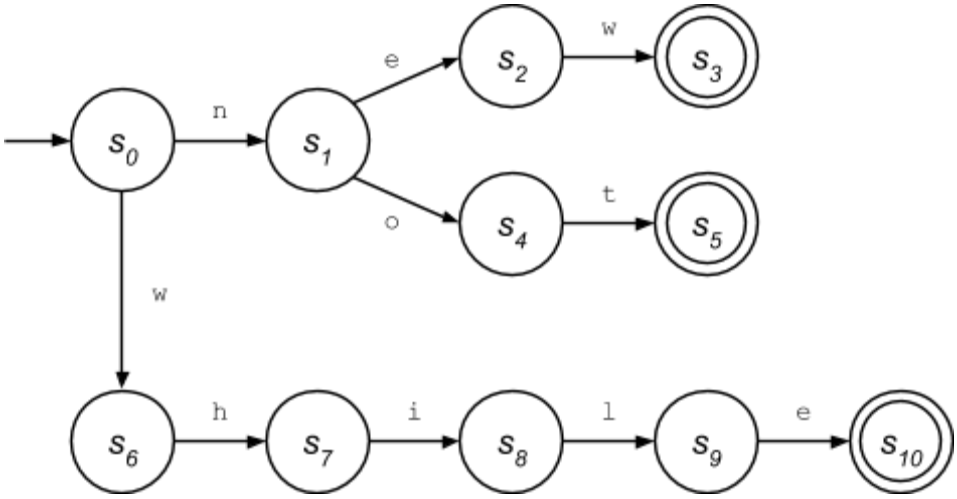
- This code can be represented with a transition diagram like this, where each node in the diagram is an abstract state in the computation (e.g. s_2 represents the state after an `'n'` and an `'e'` have been read):



- By convention, the computation starts in s_0. The state s_3 is designated as an ***accepting state*** by using a double circle. There is also an implicit error state s_e to which each state transitions on an unrecognized input.

- A similar recognizer could be built for any word by chaining together nodes into a transition diagram and then generating the corresponding if-then-else code.

- We could recognize multiple words as well by combining recognizers for individual words. For example, here's a transition diagram for a recognizer for `new`, `not`, and `while`:



- Again, the computation starts here in s_0; s_3, s_5, and s_{10} are accepting states; and there is an implicit error state s_e. We could again generate if-then-else logic corresponding to this transition diagram in a straightforward way.

- These transition diagrams are examples of a formal mathematical concept called ***finite automata***. Formally, a finite automaton (FA) is a five-tuple (S, Σ, δ, s_0, S_A), where:
  - S is the set of all states in the FA, including the error state s_e.
  - Σ is the alphabet used in the FA.
  - δ is the transition function. Specifically $\delta(s, c)$ maps the state/character pair (s, c), where $s \in S$ and $c \in \Sigma$, to another state.
    - For example, in the FA above, $\delta(s_1, \text{e}) = s_2$.
  - s_0 is the starting state, $s_0 \in S$.
  - S_A is the set of all accepting states, $S_A \subseteq S$.

- An FA accepts a given input string $c_1\ c_2\ c_3\ \ldots\ c_n$ if, starting in s_0, the FA ends up in an accepting state after making the transitions dictated by the sequence of characters.

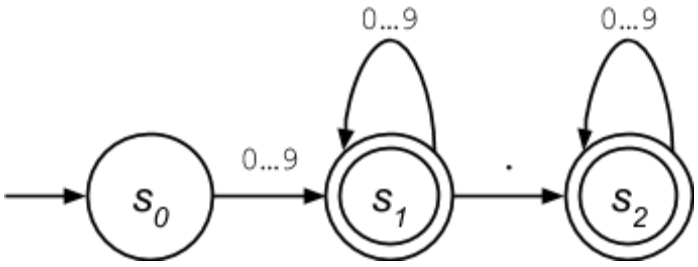
- Two error conditions are possible:
  - If some character c_j causes the FA to transition to the error state s_e, this is a lexical error. The string $c_1\ c_2\ \ldots\ c_j$ is not a valid prefix in the language accepted by the FA.
  - If the FA completely consumes the input string and ends in a non-accepting state (other than s_e), this is also an error, though the string $c_1\ c_2\ c_3\ \ldots\ c_n$ is a valid prefix of some word accepted by the FA.

- So far, so good. We can use the above model to recognize an arbitrary number of fully-specified words, such as the keywords of a language. What if we want to recognize *classes* of words, such as numbers or identifiers?

- Importantly, such classes contain words of arbitrary length, which means that an FA for these classes would potentially need to recognize an infinite number of words.

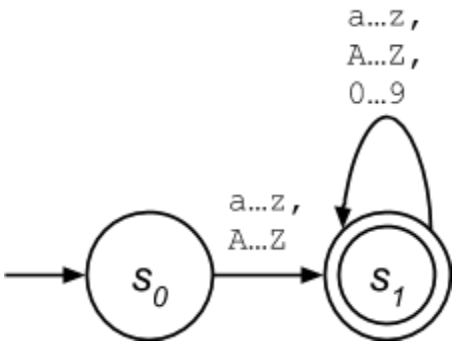
- How can we do this while keeping the FA itself finite? The key is to introduce cycles into the transition diagram.

- For example, here is a simple FA to recognize unsigned floating-point numbers with at least one leading digit before the decimal point:



- In particular, this FA captures the property that an unsigned floating-point number consists of at least one numeric character, optionally followed by a decimal point and then zero or more additional numeric character.

- Similarly, the FA below recognizes identifiers consisting of an alphabetic character followed by zero or more alphanumeric characters, similar to those used in languages like C and C++ (it would be easy to extend this definition to include underscores, etc.):



- Importantly, FAs like the ones above for recognizing unsigned floating-point numbers and identifiers raise the difference between a class of words in a language and the actual text of any one specific word.

- An entire class of words, such as floating-point numbers or identifiers, corresponds to a ***syntactic category***, i.e. the "part of speech" for those words.

- The text of any one specific word, on the other hand, is known as a ***lexeme***. The identifier `kittyCat2`, for example, is a lexeme.

Specifying FAs using regular expressions

- The set of words accepted by an FA forms a language, and our ultimate goal is to be able to use FAs to recognize this language within the compiler's scanner. To be able to do this, we need a compact and intuitive way to represent an FA.

- Both the transition diagram for the FA and the formal 5-tuple specification of the FA specify this language precisely. Unfortunately, both of these representations would be cumbersome ways for a human to specify an FA for use in a scanner.

- Fortunately, the language recognized by any FA can be described using a notation called a ***regular expression*** (or ***RE***, or ***regex***). In fact, as we'll see, regular expressions are equivalent to FAs.

- RE is a notation for defining a language. Any language that can be defined using an RE is called a ***regular language***.

- Three basic operations can be used in an RE:
    - ***Concatenation*** – for two sets of strings R and S, the concatenation of R and S is the set of all strings formed by appending an element of S onto the end of an element of R, i.e. $\{ xy : x \in R \text{ and } y \in S \}$.
        - The concatenation of these two sets is denoted RS.
    - ***Alternation*** (or ***union***) – for two sets of strings R and S, the alternation of R and S is simply the union of the two sets, i.e. $\{ x : x \in R \text{ or } x \in S \}$.
        - The alternation of these two sets is denoted $R \mid S$.
    - ***Kleene closure*** (or just ***closure***) – for a set of strings R, the Kleene closure of R is the union of the concatenation of R with itself zero or more times, i.e. $U_{i=0\ldots\infty} R^i$.
        - The Kleene closure of R is denoted R^*.

- Using these three operations, the set of REs over an alphabet Σ can be defined as follows:
    - For any character $c \in \Sigma$, *c* is an RE denoting the set containing only *c*.
    - If *r* and *s* are REs respectively representing the languages *L(r)* and *L(s)*, then:
        - *rs* is an RE denoting the concatenation *L(r)* and *L(s)*.
        - *r* | *s* is an RE denoting the union of *L(r)* and *L(s)*.
        - *r** is an RE denoting the Kleene closure of *L(r)*.
    - ϵ is an RE denoting the set containing only the empty string.

- When specifying REs, we can also make use of several widely-used shorthand notations for convenience:
    - ***Finite closure*** – for an RE *r* and a positive integer *i*, r^i is a finite closure of *r* and denotes from 1 to *i* occurrences of *r*.
        - For example r^3 is shorthand for *(r | rr | rrr)*.
    - ***Positive closure*** – for an RE *r*, r^+ is the positive closure of *r* and denotes 1 or more occurences of *r*.
        - This is shorthand for *rr**.
    - ***Range*** – for some set of characters $c_1c_2c_3\ldots c_k$ that are contiguous in the alphabet Σ, $[c_1\ldots c_k]$ denotes the union of the individual characters between and including c_1 and c_k, i.e. $(c_1 \mid c_2 \mid c_3 \mid \ldots \mid c_k)$
        - For example `[0…9]` denotes `(0|1|2|3|4|5|6|7|8|9)`.
    - ***Complement*** – for some character *c*, ^*c* denotes the complement of *c* with respect to the alphabet Σ, i.e. the set $\{\Sigma - c\}$.

- Applying these rules, we can generate the following examples:
    - `new` is an RE that recognizes only the word `new`.
    - `n(ew|ot)` is an RE that recognizes the words `new` and `not`.
        - Parentheses take highest precedence in an RE's specification.
    - `[0…9]`$^+$`(`ϵ`|.[0…9]*)` is an RE that recognizes unsigned floating-point numbers, as defined by the FA above.
    - `([a…z]|[A…Z])([a…z]|[A…Z][0…9])*` is an RE that recognizes identifiers, as defined by the FA above.
    - `"(^("|\n))*"` is an RE denoting a correctly-quoted character string that does not span multiple lines.
        - Here, `\n` is an escape sequence for a newline character.
    - `//(^\n)*\n` is an RE that recognizes C++-style comments, which begin with the delimiter `//` and continue to the end of the current line.