Regular Expression-Based Scanners

- We've seen how regular expressions (REs) can be used to specify the microsyntax for a programming language. We need now to be able to convert a collection of REs specifying a language's microsyntax into a working scanner for that language.

- At a high level, our approach to this problem involves converting the collection of REs into a ***nondeterministic finite automaton*** (***NFA***), converting the NFA into a ***deterministic finite automaton*** (***DFA***), and finally converting that DFA into executable code.

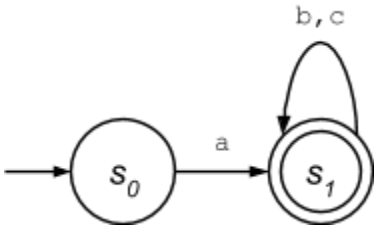
- The process for converting an RE into an NFA and then converting that NFA into a DFA is material that should have been covered in your previous coursework, so we won't specifically explore it here. The Appendix contains more about that process if you need a refresher. Here, we'll assume we have a DFA representing an RE and explore how to use that DFA to produce a working scanner.

Using a DFA to produce a recognizer

- Once we have a DFA specification, we can use it to produce an executable recognizer for the words in a language.

- Remember, a DFA consists of a 5-tuple (S, Σ, δ, s_0, S_A), where:
  - S – the set of all states in the FA, including the error state s_e
  - Σ – the alphabet used in the FA
  - δ – the transition function
  - s_0 – the starting state
  - S_A – the set of all accepting states

- DFAs are easily visualized. For example, a DFA representing the RE `a(b|c)*` looks like this (a derivation of this DFA is described in the Appendix):



- In a program (like a scanner), a DFA is typically represented in tabular form. For example, we could represent the DFA above for our language `a(b|c)*` with a single transition table, as follows:

δ	a	b	c	other
s_0	s_1	s_e	s_e	s_e
s_1	s_e	s_1	s_1	s_e

- Given such a table, we could implement a function like the following to recognize words in the language, assuming the availability of a function `nextChar()` that returns the next character in the input stream and denoting the starting state s_0 as `s_0`, the error state s_e as `s_e`, and the set of accepting states S_A as `S_A`:

```
s = s_0
c = nextChar()

while (s != s_e && c != eof)
     s = δ[s, c]
     c = nextChar()

if (s in S_A)
     return SUCCESS
else
     return INVALID
```

Implementing a full-blown scanner

- There are many details we need to flesh out in order to implement a full-blown scanner from the pieces above, including:

- How to recognize many different syntactic categories.
- How to perform scanning when the source code contains many words.
- How to deal with whitespace, comments, etc., which should be ignored in the source code.
- How to cope with potentially large transition tables representing the DFA.

Recognizing many syntactic categories

- First, we have been dealing so far with very simple languages that can be represented with a single RE. What happens when we're dealing with a real programming language, for which we might need many REs to represent the various syntactic categories?

- For example, say we have a set of REs, r_1, r_2, …, r_n, each designed to recognize a specific syntactic category in a language we're generating a scanner for.

- First, we can combine these several REs into a single RE ($r_1 \mid r_2 \mid \ldots \mid r_n$). This single RE could then be used to generate a DFA for our entire language.

- Importantly, in such a setup, extra steps would have to be taken to make sure the NFA and resulting DFA could not only recognize valid strings in the language but could also assign a syntactic category to each valid word.

- For example, if each of the original REs for the language was labeled with the specific syntactic category it represented, this information could be passed into the NFA and DFA construction procedures, so that at the end, the accepting states in the resulting DFA could be labeled with the specific syntactic categories to which they correspond.

- Typically, an additional table, which we can refer to as `Category`, is generated to encode the correspondence between states in the final DFA and syntactic categories in the language, e.g.:

State	Category
s_0	`invalid`
s_1	`identifier`
s_2	`invalid`

s_3	`number`
s_4	`=`
...	...

- If multiple different categories are accepted by a particular state (e.g. the string `new` might match the RE for identifiers as well as an RE for keywords), then a priority ordering can be specified so that if multiple categories are matched, the one with highest priority is assigned.

Handling multiple words in the input stream

- Up to now, we've primarily focused on determining whether or not a single word belongs to a particular language specified by an RE. Typical source code, however, contains many words.

- Fortunately, we can adapt the mechanisms we employed above in a straightforward manner to deal with an input stream of many words.

- In particular, we can adapt the DFA so that it reads the input stream until the current state has no valid transition on the next character.

- At this point, there are several possibilities:
    - If the current state is an accepting state, the scanner can assign the current word a syntactic category (e.g. using the `Category` table), reset the current state to s_0, and reset the current word to the empty string.
    - If the current state is not an accepting state but the scanner encountered an accepting state at some point during its formation of the current word, the scanner can "roll back" to the last encountered accepting state, replacing characters from the current word back into the input stream as it does so. Again, the scanner can reset itself to prepare to recognize the next word.
    - If the scanner did not encounter an accepting state during the formation of the current word, it can report an error. It can optionally reset itself in order to continue scanning, with the goal of identifying as many invalid words as possible.

Ignoring whitespace, comments, etc.

- A final detail we need to deal with is how to handle parts of the source code that don't belong to words in the program text, e.g. whitespace or comments, which are typically ignored.

- To handle these kinds of strings, additional REs are typically included to specify patterns that should be ignored in the source code.

Compressing the DFA transition table

- For a typical source language, the transition table representing the DFA for the language's recognizer can grow quite large. In order to reduce the memory footprint required by the scanner, it is possible to compress this table by combining characters into character classes.

- This is usually done by grouping characters whose transition table columns are identical into a single class.

- For example, in the transition table above for the language `a(b|c)*`, the characters `b` and `c` have identical columns in the transition table and can be combined into a single character class, resulting in the elimination of one column from the transition table:

δ	a	b,c	other
s_0	s_1	s_e	s_e
s_1	s_e	s_1	s_e

- For most real languages, such grouping can result in significant space savings.

- In order to encode information about character classes, an additional table, which we can call `CharClass`, is typically generated. For the simple example above, the `CharClass` table might look like this:

character	class
a	a
b	b,c
c	b,c
other	other

Table-driven scanners

- At this point, we can combine all of the concepts discussed above to generate a type of scanner called a table-driven scanner.

- Specifically, given a set of REs describing the microsyntax for a language, we can use the procedures described above to generate the tables δ, `Category`, and `CharClass`.

- Once these tables are generated for our source language, they can be used to create a function `nextWord()`, which will return both the spelling and syntactic category of the next word in the input stream:

```
nextWord()
     state = s_0
     word = ""
     stack = newStack()

     while (state != s_e)
          c = nextChar()
          word += c
          if (state in S_A)
               clearStack(stack)
          push(state, stack)
          class = CharClass(c)
          state = δ(state, class)

     while (!(state in S_A) && !isEmpty(stack))
          state = pop(stack)
          c = truncate(word)
```

```
            rollBack(c)

        return (word, Category[state])
```

- Importantly, `nextWord()` assumes the existence of the following:
    - An implementation of a stack data structure, used to help roll back if needed.
    - A function `truncate()`, which removes and returns the last character of a string.
    - A function `rollBack()`, which replaces a character into the input stream.

- The most important parts of `nextWord()` are the two loops.

- The first loop executes the DFA, reading one character at a time and updating the current state based on the transition table (compressed as described above).
    - This loop also pushes each state it sees onto a stack, in order to allow the scanner to roll back in case it overshoots the end of a valid word.

- The second loop performs this rollback, popping states off of the stack until it rolls back to the last encountered accepting state.
    - Each time a state is popped from the stack, the corresponding character is removed from the end of the current word and added back into the input stream.

- At the end of the function, a pair containing the spelling of the current word and its syntactic category is returned.

- To perform scanning, the `nextWord()` function can be called repeatedly until the input stream is exhausted.

- Many optimizations are possible here, for example to reduce the amount of rollback required or to reduce the cost of computing DFA transitions. However, a table-driven scanner like the one above is complete and usable in practice.

- Other non-generated scanners are also possible. For example, many popular compilers like gcc use hand-coded scanners in which the RE-based specification is foregone in favor of encoding the language's microsyntax directly in the implementation of the scanner.

Appendix: Equivalence of REs and DFAs

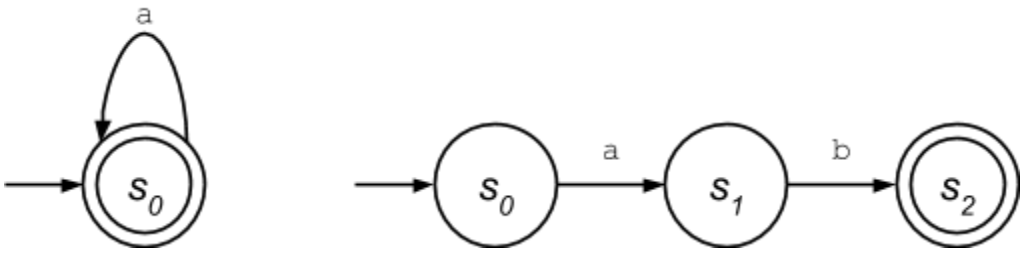
Here, we provide more details on the equivalence between regular expressions and deterministic finite automata, including a method for converting an RE into a DFA.

Nondeterministic finite automata

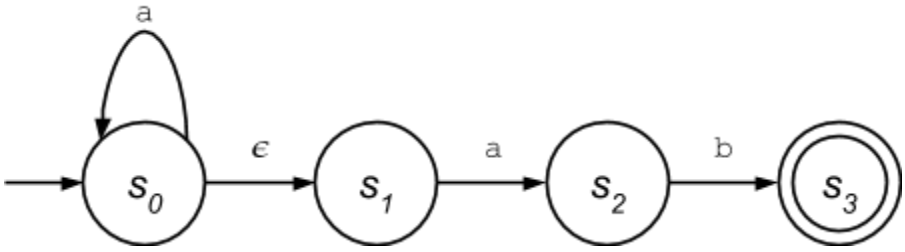
- A nondeterministic finite automaton (NFA) is a finite automaton that allows transitions on the empty string ϵ. Such transitions are called ϵ-transitions.

- ϵ-transitions are particularly useful because they make it easy to combine FAs to form more complex FAs.

- For example, consider the following FAs that recognize the languages `a*` and `ab`:



- We could use an ϵ-transition to merge these FAs into a single FA to recognize the language `a*ab`:



- In effect, this ϵ-transition allows the FA to make two different transitions out of state s_0 on character `a`: one back to s_0 and one to s_2.

- Under the NFA model, the finite automaton can choose *nondeterministically* between these transitions to recognize different strings from the language `a*ab`.

- For example, on the string `ab`, the FA would immediately choose to take the ϵ-transition to s_1, after which it would transition to s_2 on the character `a` and then to s_3 on the character `b`.

- On the string `aab`, on the other hand, the FA would first choose to transition back to s_0 on the character `a`, after which it would choose to take the ϵ-transition to s_1, followed by a transition to s_2 on the character `a` and then to s_3 on the character `b`.

- In general we can view the behavior of an NFA on a particular string in two different ways. Specifically, each time an NFA must make a nondeterministic choice between transitions, it can do one of the following two things:
    - Omnisciently follow the transition that leads to an accepting state for the input string, if one exists.
    - Make a clone of itself to pursue each possible transition simultaneously. Under this model, the NFA accepts a string if one or more clones ends up in an accepting state.

- In contrast to an NFA, a finite automaton that has a unique transitions out of every state for each character in the language is called a deterministic finite automaton (DFA).

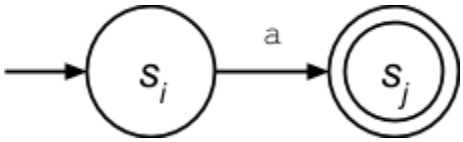
- Importantly, it can be proven that DFAs and NFAs are equivalent. This fact will be useful later.

RE to NFA with Thompson's construction

- As we described above, the first step in moving from a collection of REs describing a language's microsyntax to a scanner for that language is to convert the REs into an NFA.
    - In your Theory of Computation course, you should have seen that regular expressions and NFAs are equivalent.

- To do this, we can use Thompson's construction, which provides a straightforward set of template transformations on NFAs to model the result of each of the basic RE operators (concatenation, union, and closure).

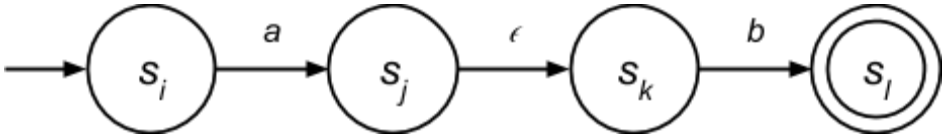
- The first piece of Thompson's construction is an NFA that recognizes the RE for a single character. For example, the NFA for the RE `a` is below:



- Next, we'll see template transformations for each of the basic RE operations. Each of these transformations describes how to take NFAs representing REs and combine or augment them into an NFA that applies one of the RE operations to the original REs.

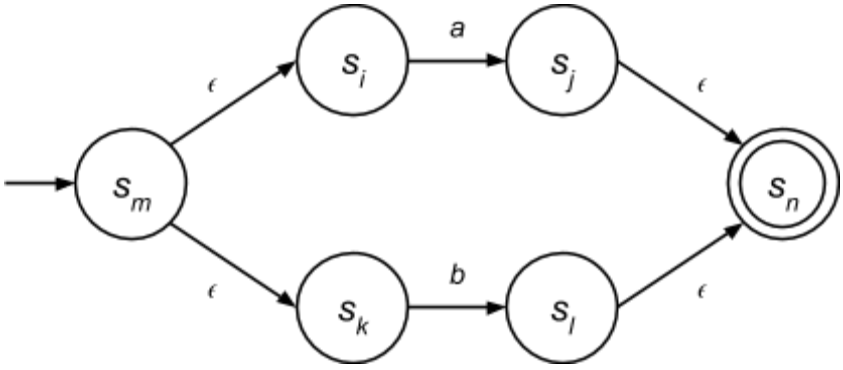
- The NFAs that serve as inputs to these transformations can either be simple NFAs for one-character REs, like the one just above, or they can themselves be NFAs that were generated by a previous transformation.

- The transformation to form an NFA that representing the concatenation *ab* of two REs *a* and *b* is:



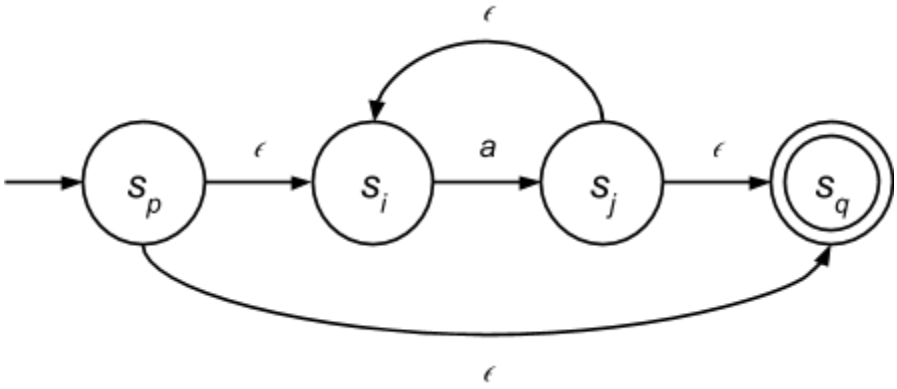
- Here, s_i is the start state for the RE *a*, and s_j is the accepting state for that RE. Similarly, s_k is the start state for the RE *b*, and s_l is that RE's accepting state.

- The transformation to form an NFA representing the union *a* | *b* of two REs *a* and *b* is:

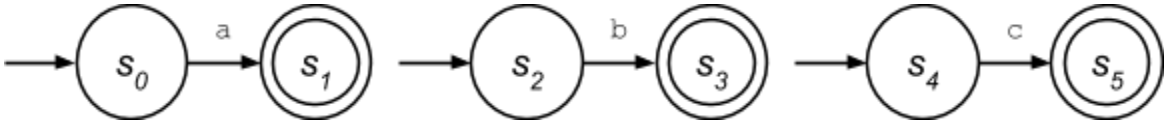


- Here again, s_i is the start state for the RE *a*, s_j is the accepting state for that RE, s_k is the start state for the RE *b*, and s_l is that RE's accepting state.

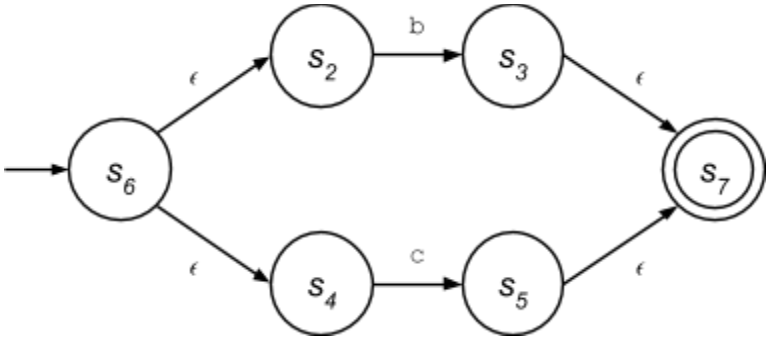
- Finally, the transformation to form an NFA representing the closure *a** of an RE *a* is:



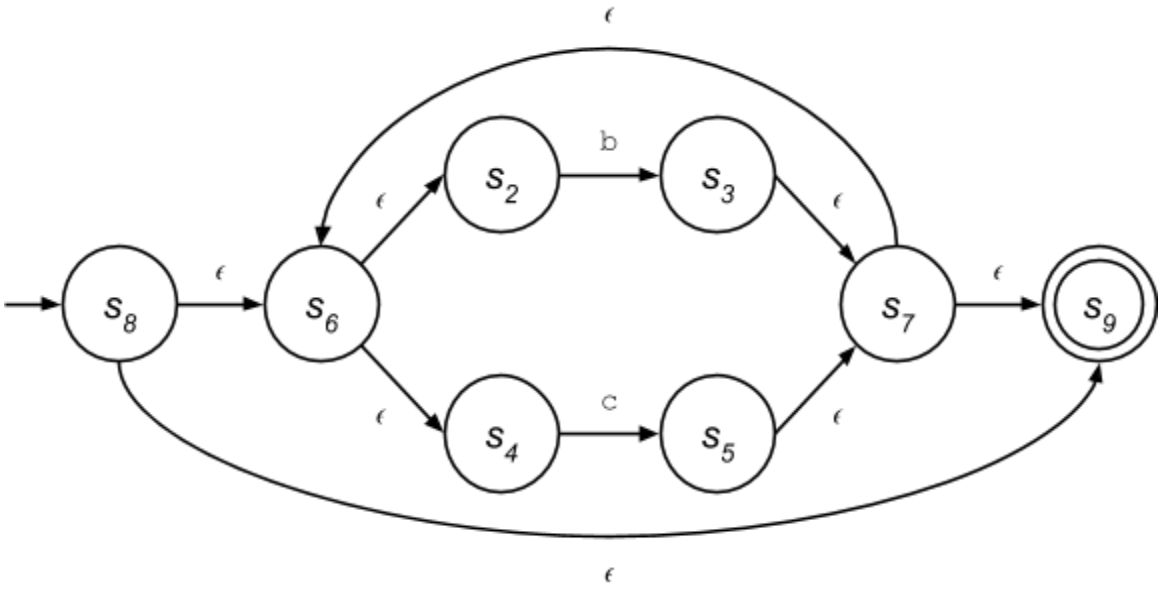
- As an example of how these transformations can be applied, let's build an NFA for the RE `a(b|c)*`. To start, here are the NFAs for the REs recognizing the individual characters `a`, `b`, and `c`:



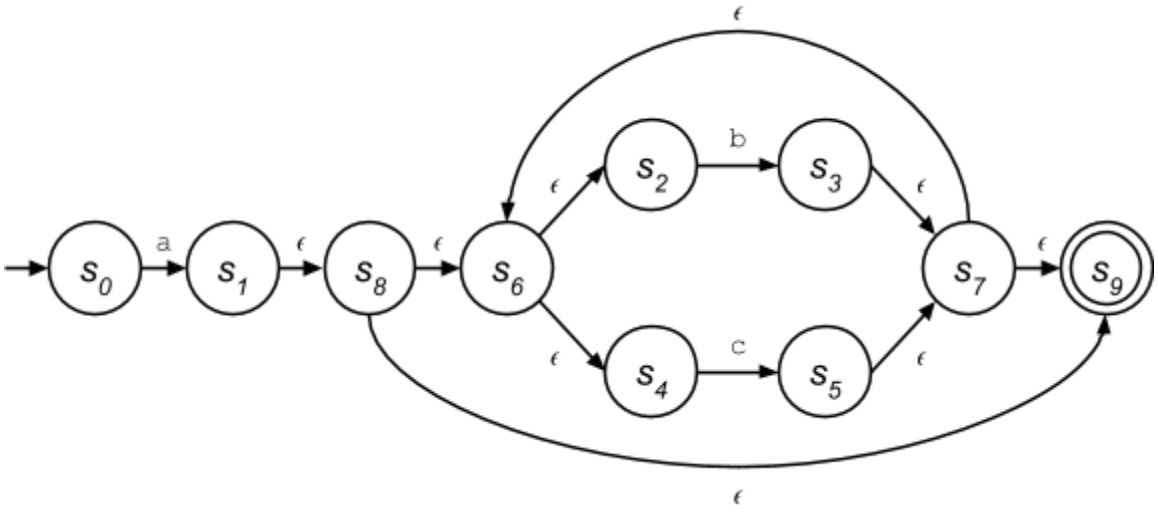
- Next, here is how the transform representing the concatenation operation can be applied to combine the NFAs for `b` and `c` to form an NFA for `b|c`:



- Next, here is how the transform representing the closure operation can be applied to form an NFA for `(b|c)*`:



- Finally, here is how the transform representing the concatenation operation can be applied to the NFAs for `a` and `(b|c)*` to form an NFA for `a(b|c)*`:



NFA to minimal DFA with the subset construction and Hopcroft's algorithm

- Thompson's construction gives us a way convert an RE into a computational specification in the form of an NFA.

- However, execution of an NFA is much more challenging than execution of a DFA. Thus the next step of our process for transforming a set of REs into a scanner is to convert the NFA output by Thompson's construction into a DFA.
    - Note that to reduce the computational expense of executing our DFA, we will want to produce a *minimal* DFA.

- To do this, we will first apply the subset construction to convert the NFA into a DFA. Then, we will apply Hopcroft's algorithm to minimize this DFA.

- You should have covered the subset construction in your Theory of Computation course, so we won't go into it in detail here.
    - The textbook for our course describes the subset construction in sufficient detail if you'd like to review.

- At a high level, the subset construction builds a set of ***configurations***, where each configuration represents the *set* of states in which the NFA could possibly be after seeing a particular string of characters.
- A new configuration q_j is formed from a previous configuration q_i by computing all of the states that could be reached from the states in q_i after seeing a particular character.
    - For example, in the NFA above, we would have an initial configuration q_0 containing only the state s_0. After seeing a character `a` in s_0, the NFA could possibly be in the following states, depending on what ϵ-transitions it followed: s_1, s_8, s_6, s_9, s_2, s_4.
    - These 6 states would be grouped together into a separate configuration, say q_1.
    - Additional configurations could be formed by computing all of the states possible after seeing a character `b` in any of the states in configuration q_1, after seeing a character `c` in any of the states in configuration q_1, etc.

- At the end of the construction, each configuration corresponds to a state in the DFA. If a particular configuration contains an accepting state in the NFA, then the DFA state corresponding to that configuration becomes an accepting state. The transition function of the DFA is formed by observing all possible transitions between configurations.

- After we use the subset construction to generate a DFA from our NFA, we can use Hopcroft's algorithm to minimize the number of states in the DFA.

- Conceptually, Hopcroft's algorithm finds states in the DFA that have the same behavior on every input character in the alphabet. When such identically-behaved states are found, they are grouped together into a single state.
    - More details on Hopcroft's algorithm are available in our course textbook.

- After running Hopcroft's algorithm, we will have a minimal DFA to serve as the basis for an executable scanner.

- For example, the minimal DFA for our language `a(b|c)*` looks like this: