7 Scanning and regular languages

In the course so far, we started with machine language and built various language abstractions on top of it. Although we can now program in terms of these abstractions, our programs expressed as Scala `Code` data structures are difficult to read and write, and there is little error checking to ensure that we use the abstractions correctly. In the next part of the course, we will shift from *adding more abstractions* to *improving the usability of the language*. Specifically, we will learn how to translate a program in Lacs, a language that looks like Scala or other high-level programming languages, into the corresponding `Code` data structure.

7.1 Formal languages

The practical implementation of this part of our compiler will be grounded in the theory of *formal languages*, which describes sets of strings with mathematical rigour. In our applications, the strings are programs or parts of programs. We will define the sets of strings that are valid programs in a particular programming language. Formal language theory also makes it possible to automatically derive an implementation of parts of the compiler from the language specification. Besides saving us implementation work, this approach helps to ensure that the language that the compiler implements is the same as the language that we specify.

We begin by defining some terminology that applies to formal languages in general. An *alphabet* is any *finite* set of *symbols*. Some examples of alphabets are: the set of binary digits $\{0, 1\}$, the set of decimal digits $\{0, \ldots, 9\}$, and the set of uppercase and lowercase letters $\{A, \ldots, Z, a, \ldots, z\}$. In some contexts, we might define the *alphabet* to be a set of things that we would more informally consider to be words, such as the set of Scala keywords: $\{\texttt{def}, \texttt{val}, \texttt{var}, \ldots\}$. However, in this case, these are formally not *words*, but the *symbols* of the *alphabet*. It is common to use the letter Σ to denote an alphabet.

A *word* over some given alphabet is a *finite* sequence of symbols drawn from that alphabet. For example, 123 is a word over the alphabet of decimal digits and *abc* is a word over the alphabet of letters. A word can consist of any number of symbols, including zero symbols. A word with zero symbols is called the empty word. We usually write the empty word as ε, because if we simply wrote nothing (no symbols), we would not be able to see the word. However, ε is not considered a *symbol* in the alphabet, so ε is not a word with one symbol ε, but a notation for the word with zero symbols.

A (formal) *language* is a *set* of words over some alphabet, in the mathematical sense of the word set. Some examples are:

- the set of binary strings of length 32 (i.e. MIPS words)
- the set of prime numbers when written in decimal
- the set of strings of letters that begin with the letter q
- the set of valid Scala programs
- the set of correct solutions to Assignment 6

- the empty set containing no words
- the set containing exactly one word, the word ε

Finally, we can classify a *language* into one of a number of *language classes* depending on properties of the language. Some examples of relevant language classes are:

- *Finite languages* are *finite* sets of words.
- *Regular languages* are sets of words that can be specified using the techniques that we will learn in this module.
- *Context-free languages* are sets of words that can be specified using more powerful techniques that we will learn in the next module.
- *Decidable languages* are sets of words for which there exists an algorithm that decides membership, i.e., decides whether or not some given word is in the set. Decidable and undecidable languages are studied in depth in CS 341.

In practice, in our compiler, we will use a *regular language* for the process of *scanning*, which means to partition an input program, a string of characters, into a sequence of keywords, symbols, identifiers, and literals, collectively called *tokens*. For example, the Scala (or Lacs) program

```
def main(foo:Int,bar:Int)={foo+bar}
```

will be scanned as:

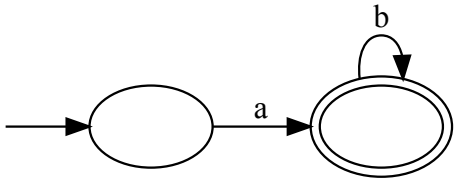
`def` `main` `(` `foo` `:` `Int` `,` `bar` `:` `Int` `)` `=` `{` `foo` `+` `bar` `}`

We will use a *context-free language* to recognize the tree structure of the program, for example to determine that one procedure is nested inside another or that `b*c` is a subexpression of the larger expression `a+b*c+d`. This process is called *parsing*.

In general, we will want to use *decidable languages* to specify valid Lacs programs, since we want it to be possible for our compiler to determine whether a particular input program is valid. All *regular* and *context-free* languages are *decidable*.

7.2 Deterministic finite automata

To specify the scanning process, we will use *deterministic finite automata (DFAs)*. A DFA is a directed graph whose nodes are called *states* and whose edges are called *transitions*. Each transition is labelled with a symbol from the alphabet Σ. One state of the DFA is identified as the *start state*. Some subset of the states are *accepting states* (also called *final states*). In a given state and for a given symbol, there can be at most one transition that originates from the state and is labelled with that symbol; multiple transitions originating from the same state and labelled with the same symbol are not allowed. The following diagram shows an example of a DFA over the alphabet $\Sigma = \{a, b\}$:



The start state is the state on the left. In general, in DFA diagrams, the start state is identified by an arrow pointing into it. The state on the right is an accepting state. In general, in DFA diagrams, accepting states are drawn with a double border. The arrows in the diagram represent transitions. They are labelled with symbols from the alphabet.

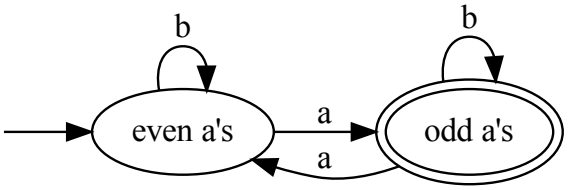
Each DFA is a specification of a *language*, a set of *words* over the *alphabet*. Informally, we can determine whether a word $w = w_1 w_2 \dots w_n$ is in the language specified by the DFA using the following process. We start in the *start state* of the DFA and consider the symbols w_i one at a time, in order. For each symbol, we follow a transition labelled with that symbol from the current state to a new state of the DFA. If there is no transition labelled with the symbol, the DFA is said to *get stuck* and the word is not in the language. If the DFA goes through the whole word without getting stuck and ends up in an *accepting state*, then the word is in the language. If the DFA ends in a *non-accepting* state after processing the whole word, then the word is not in the language.

The *language* specified by the example DFA above is the set of all words that start with one a, which is then followed by zero or more bs:

$$L = \{a, ab, abb, abbb, \dots\}$$

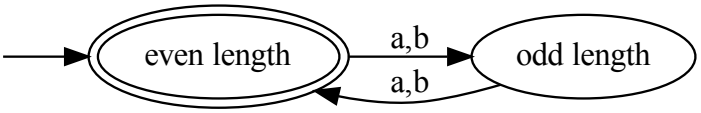
Using any word in this set, we can follow the transitions and end up in the state on the right, which is accepting. Given a word such as aa, however, we would transition from the left state to the right state on the first a, but then there is no transition out of the right state labelled a, so the DFA would get stuck. Given the empty word, the DFA would start and immediately end in the left state, which is not accepting. Therefore, neither aa nor the empty word are in the language specified by this DFA.

As another example, the following DFA specifies the language of words over the alphabet $\Sigma = \{a, b\}$ that contain an odd number of as (and any number of bs):



It is common to give a name to each state to describe the sets of words that send the DFA to that state, in this case words with an even or odd number of as. When designing a DFA to specify a particular language, one practical approach is to first think about and decide which states will be needed to distinguish different sets of words relevant to the desired language, and then to connect the states with appropriate transitions. In this example, a word has either an even or an odd number of as. Whenever the DFA sees an a, the parity of the as changes between even and odd; whenever it sees a b, the parity of the bs does not change.

As one more example, the following DFA specifies the language of words of even length over the alphabet $\Sigma = \{a, b\}$.



In this case, we use the shorthand notation a, b to indicate that each arrow in the diagram actually represents two transitions, one labelled a and the other labelled b.

7.3 DFAs formally

The graphical notation for DFAs is easy to draw and interpret. We can formalize the same concept so that we can talk about DFAs precisely. A *deterministic finite automaton (DFA)* is a five-tuple $\langle \Sigma, Q, q_0, A, \delta \rangle$ where:

- Σ is a finite set of *symbols* (the *alphabet*).
- Q is a finite set of *states*.
- $q_0 \in Q$ is the *start state*.
- $A \subseteq Q$ is the set of *accepting states*.
- $\delta : Q \times \Sigma \to Q$ is the *transition function* that, given a state q and alphabet symbol a, returns the new state q' to transition to after processing the symbol a in state q.

For example, we can formally write the above example DFA specifying words of even length as: $M = \langle \{a, b\}, \{even, odd\}, even, \{even\}, \delta \rangle$, where

$$\begin{aligned} \delta(even, a) &= odd \\ \delta(even, b) &= odd \\ \delta(odd, a) &= even \\ \delta(odd, b) &= even \end{aligned}$$

Now that we have this formal notation, we can formally define the language specified by a DFA. Given a DFA $\langle \Sigma, Q, q_0, A, \delta \rangle$, we first define the *extended transition function*

$\delta^* : Q \times \Sigma^* \to Q$, which maps a state q and a *sequence* of symbols w to a new state q' as follows:

$$\delta^*(q, \varepsilon) = q$$
$$\delta^*(q, \mathit{first} :: \mathit{rest}) = \delta^*(\delta(q, \mathit{first}), \mathit{rest})$$

The intuition is that if we start in DFA state q and follow transitions according to the symbols in the sequence w, then we will end up in the state given by the extended transition function $\delta^*(q, w)$. When w is the empty sequence ε, δ^* just returns the state q that we start in. When w is non-empty and consists of one symbol *first* followed by the rest of the sequence *rest*, we apply δ to *first* to make the first transition in the DFA on the symbol *first*, then recursively apply δ^* to make additional transitions on the symbols in the rest of the sequence *rest*.

Now that we can make transitions through the DFA on a *sequence* of symbols rather than only a single symbol at a time, we say that the DFA *accepts* a word w if $\delta^*(q_0, w) \in A$. In words, if we start in the start state q_0 and make transitions according to the symbols in w, the state that we end up in is an accepting state. We then define the *language specified by the DFA* as the set of words that the DFA *accepts*.

Finally, we can define a regular language. A language (i.e., a set of words) is *regular* if there exists a DFA that specifies it. All of the languages specified by the example DFAs that we have discussed here are regular.

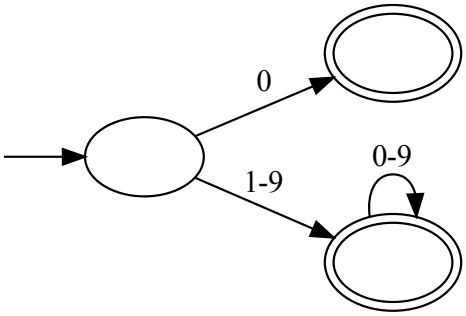
Exercise 18. *Convince yourself that every* finite *language is also* regular. *Given a finite set of words, think about how you would create a DFA that specifies that set. On the other hand, use a counterexample to argue that not every* regular *language is* finite.

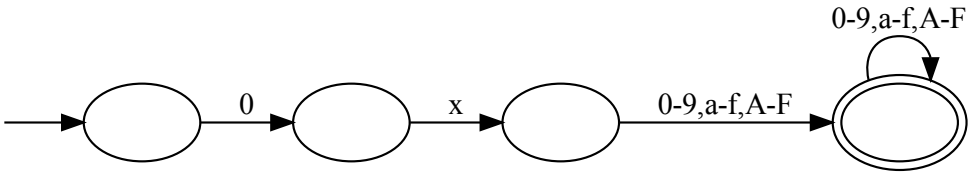
For a given language, there may be many different DFAs that specify it. However, the minimal DFA (i.e., the one with the smallest number of states) specifying a given language is unique, as you can prove in CS 462.

7.4 Non-deterministic finite automata

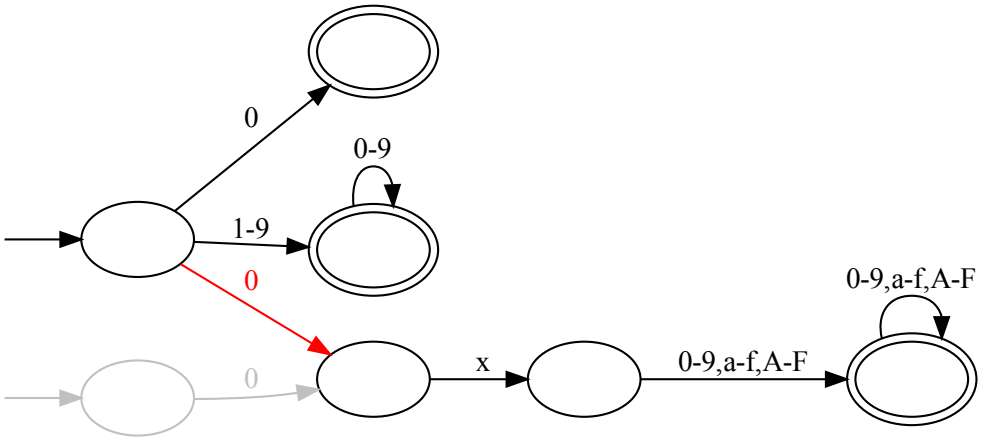
Consider the following two DFAs that specify:

- the set of natural numbers written in decimal with no leading zeros and
- the set of natural numbers written in hexadecimal, prefixed with `0x`, as in C, C++, Java, or Scala.





Suppose we wanted to combine these DFAs to specify the union of their languages, the set of natural numbers written either in decimal or in hexadecimal. We might try to combine the DFAs as follows by adding the transition shown in red and removing the state and transition shown in grey:



The resulting automaton does accept words from both languages, but it is no longer *deterministic*: in the start state on the symbol `0`, *two* transitions are possible (the red one and the black one) and it is not obvious which one to take. Nevertheless, such a

non-deterministic finite automaton (NFA) can also be used to specify a language. For a given word w, there may be multiple possible paths through an NFA that end in different states. In the example NFA above, on the input word $w = \texttt{0}$, the NFA may end either in the *accepting* state at the top of the diagram or in the *non-accepting* state at the end of the red transition. Should the NFA accept the word `0` or not? We define an NFA to *accept* a word w if *at least one* of the paths through the NFA following the symbols in w ends in an accepting state. Thus, this example NFA does accept the word `0`.

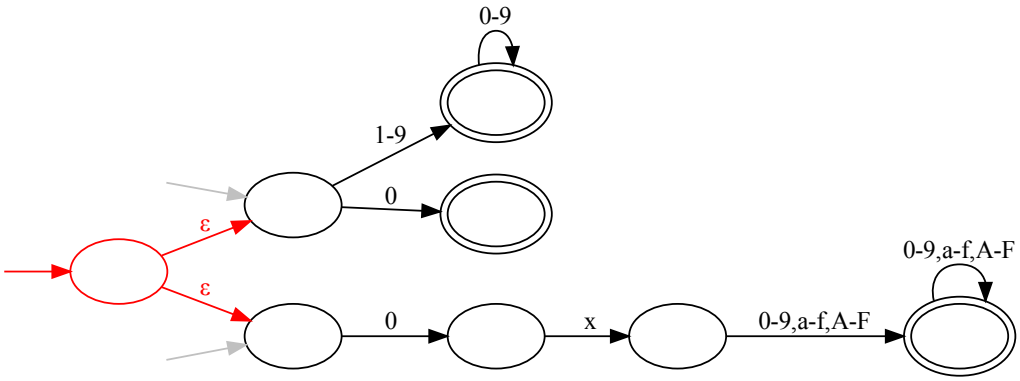
As with DFAs, we can define an NFA formally as a five-tuple $\langle \Sigma, Q, q_0, A, \delta \rangle$ whose components are the same as those of a DFA, except that the transition function $\delta : Q \times \Sigma \to 2^Q$ now maps a state q and an alphabet symbol a to a *set* of states $Q' \subseteq Q$. The notation 2^Q here denotes the set of sets of states. As with DFAs, we can define an extended transition function $\delta^* : Q \times \Sigma^* \to 2^Q$ that computes the *set* of states that the NFA can end up in if it starts in some state q and follows transitions corresponding to a *sequence* of symbols w:

$$\delta^*(q, \varepsilon) = \{q\}$$
$$\delta^*(q, \mathit{first} :: \mathit{rest}) = \bigcup_{q' \in \delta(q, \mathit{first})} \delta^*(q', \mathit{rest})$$

When the sequence consists of one symbol *first* followed by the rest of the sequence *rest*, we apply δ to *first* to obtain the set of states q' in which the NFA can be after the first step, and then we apply δ^* recursively to explore all the paths starting from each of those states q' on the rest of the sequence *rest*. The result is the union of all of those sets of possible ending states over the set of states q' after the first step. Then the NFA *accepts* a word w if $\delta^*(q_0, w) \cap A \neq \{\}$. In words, the intersection of the states that the NFA reaches on the given word w and the set of accepting states is non-empty, so there is at least one state that is both reached by the NFA and is an accepting state. The *language specified by the NFA* is the set of words that the NFA *accepts*.

7.5 ε-transitions

When we combined the DFAs for decimal and hexadecimal numbers in the previous section, we added a transition between the two DFAs, but we also had to remove a state and a transition of one of the DFAs. To make it easier to more systematically combine NFAs, we add *ε-transitions*. Whenever the NFA is in a state with an outgoing ε-transition, it may optionally choose to follow that transition without processing any symbols from the input word. Using ε-transitions, we can combine the two DFAs without having to modify them, just by adding a new start state with ε-transitions to the former start states of the two DFAs, as shown in the following diagram:



From the new start state, a path through the NFA starts by following one of the ε-transitions to one of the former start states, depending on whether the input word is a decimal or hexadecimal number.

7.6 Regular expressions

Regular expressions are one more way of specifying the same languages that can be specified by DFAs and NFAs. A regular expression over an alphabet Σ is defined according to the following rules.

- $\emptyset$ is a regular expression and its meaning is $L(\emptyset) = \{\}$, the empty set of words.
- ε is a regular expression and its meaning is $L(\varepsilon) = \{\varepsilon\}$, the set containing only the empty word.
- For each symbol $a \in \Sigma$ in the alphabet, a is a regular expression and its meaning is $L(a) = \{a\}$, the set containing only the word that consists of exactly one symbol, the symbol a.
- Whenever R_1 and R_2 are regular expressions, $R_1|R_2$ is also a regular expression and its meaning is $L(R_1|R_2) = L(R_1) \cup L(R_2)$, the union of the sets of words denoted by R_1 and R_2.
- Whenever R_1 and R_2 are regular expressions, R_1R_2 is also a regular expression and its meaning is $L(R_1R_2) = \{xy \mid x \in L(R_1), y \in L(R_2)\}$, the set of words that can be split into two parts x and y such that x is in the set denoted by R_1 and y is in the set denoted by R_2.
- Whenever R is a regular expression, R^* is also a regular expression and its meaning is $L(R^*) = \{x_1x_2 \ldots x_n \mid n \geq 0, \forall i.x_i \in L(R)\}$, the set of words that can be split into zero or more parts $x_1, x_2, \ldots, x_n$ such that each of the parts x_i is in the set denoted by R.

The following are some examples of regular expressions and their meanings.

- $L(a^*) = \{\varepsilon, a, aa, aaa, \dots\}$, the set of words of 0 or more as.
- $L(aa^*) = \{a, aa, aaa, \dots\}$, the set of words of 1 or more as.
- $L((a|b)^*) = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, \dots\}$, the set of words containing 0 or more as and bs in arbitrary order.
- $L(a^*|b^*) = \{\varepsilon, a, aa, aaa, \dots, b, bb, bbb, \dots\}$, the set of words *either* of 0 or more as *or* of 0 or more bs, but without mixing as and bs.
- $L((aa)^*) = \{\varepsilon, aa, aaaa, aaaaaa, \dots\}$, the set of words of an even number of as.
- $L((a|b)^*aa(a|b)^*)$ is the set of words of as and bs that contain two consecutive as.
- $L((a|\varepsilon)(b^*b(a|\varepsilon))^*)$ is the set of words of as and bs that do not contain two consecutive as.

Kleene's theorem tells us that DFAs, NFAs with and without ε-transitions, and regular expressions all have the same expressive power: for a language L, the following statements are equivalent:

1. There exists a DFA that specifies L.
2. There exists an NFA without ε-transitions that specifies L.
3. There exists an NFA with ε-transitions that specifies L.
4. There exists a regular expression that specifies L.

Languages that can be specified by a DFA, NFA, or regular expression are called *regular languages*.

Exercise 19. *You will prove Kleene's theorem in CS 360 or CS 365, but you can think now about how to approach proving it and you can try to prove parts of it. In general, you will need to prove that each statement i in the theorem above implies statement $i-1$ of the theorem, and that statement 1 implies statement 4. For example, to prove that statement 3 implies statement 2, you need to find a way to take an NFA with ε-transitions and construct an equivalent NFA without ε-transitions that specifies the same language.*

7.7 Scanning

Since DFAs, NFAs, and regular expressions specify languages, sets of words, the obvious question associated with them is *recognition*: given a word w, is w in the specified language? In Assignment 7, you will implement a Scala function that solves the recognition problem for DFAs.

In a compiler, however, the process of *scanning* needs to answer a different question: given a word w, can w be partitioned into subwords $w = w_1 w_2 \dots w_n$ such that each subword w_i is in some given language L? Usually, this language L is a regular language specified by a DFA. In defining this language L for a particular programming language, we are specifying the forms of tokens allowed in the programming language, such as numbers and other literals,

identifiers to name variables and procedures, the keywords of the programming language, and other symbols used in programs. The job of the scanner is to partition the program, a string of characters, into substrings that are valid tokens, that is, substrings that are words in the language of valid tokens L.

In practice, tokens are annotated with a *kind*, with each accepting state in the DFA generating a particular kind of token. For example, a token `42` might be assigned the kind *number*, a token `x` might be assigned the kind *identifier* indicating a variable name, a token `def` might be assigned the kind *def* identifying this specific Scala keyword, and a token `+` might be assigned the kind *plus* identifying this specific operator. Therefore, in general, a *token* is a pair of a *kind* and a *lexeme*, where the lexeme is the actual string of characters that were scanned to make up the token, such as `42`, `x`, `def`, or `+`.

We can state the *scanning problem* precisely as follows:

- The input is a word w and a language L of valid tokens.
- The output is a sequence of non-empty words $w_1, w_2, \ldots, w_n$ such that:
  1. If we concatenate all the words in the sequence, we get back w: $w_1 w_2 \ldots w_n = w$.
  2. Each of the words is in the language of valid tokens: $\forall i. w_i \in L$.

Typically, L is a regular language.

The scanning problem does not always have a solution. For example, there is no solution when w is the word `0x` and L is the language of decimal and hexadecimal numbers specified by the NFA in a previous section. This is because `0x` $\notin L$, so `0x` cannot itself be a single token, so we are forced to split it into two tokens `0` and `x`, but `x` $\notin L$.

Deciding when the problem has a solution is quite easy: if we have a regular expression R specifying L, then the scanning problem has a solution if and only if w is in the language of a different regular expression R^*. The scanning problem corresponds exactly to the definition of the Kleene star operator * of regular expressions: $w \in L(R^*)$ exactly when we can partition w into subwords that are all in the language of R.

When a solution does exist, identifying a specific solution is less easy, and the solution is not necessarily unique. For example, when $w = aaaaa$ and $L = \{aa, aaa\}$, there are two solutions $aa \mid aaa$ and $aaa \mid aa$. As another example, when L is the language of natural numbers written in decimal, for which we saw a DFA earlier, the word $w = 42$ can be scanned as either a single token `42` or as two separate tokens `4` `2`.

A specification of a programming language should be precise and unambiguous, so that everyone can agree on the meaning of any particular program. More concretely, when we write the characters `42` in a program in Scala, for example, we probably want the compiler to interpret them as the number 42, rather than as two separate numbers 4 and 2. Therefore, it is common when specifying a programming language to modify the scanning problem as follows, into the *maximal munch scanning problem*, by adding a third condition on the output:

- The input is a word w and a language L of valid tokens.
- The output is a sequence of non-empty words $w_1, w_2, \ldots, w_n$ such that:

1. If we concatenate all the words in the sequence, we get back w: $w_1 w_2 \dots w_n = w$.
2. Each of the words is in the language of valid tokens: $\forall i. w_i \in L$.
3. Each w_i is the longest prefix of the remainder of the input $w_i w_{i+1} \dots w_n$ that is in L.

The maximal munch scanning problem requires the scanner, at each step, to greedily find the longest token possible before continuing to the rest of the input to try to find more tokens. This gives it one major advantage: its output is unique; there is only one way to scan any given input program. A disadvantage is that the maximal munch scanning problem might not have any solution in some cases where the original scanning problem does. In other words, there are input programs that can be partitioned into sequences of valid tokens, but a maximal munch scanner will not find any such sequence and will reject the program with an error. For example, when $L = \{aa, aaa\}$ and $w = aaaa$, there is a scanning solution $\boxed{aa}\boxed{aa}$ but there is no maximal munch scanning solution because the first token must be aaa, leaving only a, which is not a valid token. Another disadvantage is that when a programming language is specified with maximal munch scanning, the intuition for which programs are valid is obscured, because it is more difficult to intuitively determine whether a given input program w can be scanned using maximal munch. Despite these disadvantages, maximal munch scanning is widespread in practice because of its guarantee of a unique solution and especially a solution that prefers longer tokens such as $\boxed{42}$ over shorter tokens such as $\boxed{4}$ and $\boxed{2}$.

7.8 Maximal munch scanner implementation

One of the maximal munch scanning requirements is to find the longest prefix of the remaining input that is in the language L. When L is specified using a DFA, we can do this by first running the DFA on the remaining input until it gets stuck (or we reach the end of the input). If the DFA ends up in an accepting state, then the prefix of the remaining that we processed while running the DFA is in L by definition. If the DFA ends up in a non-accepting state, then we backtrack to the last accepting state that the DFA visited. The prefix of the remaining input that we processed while the DFA ran to that last-visited accepting state is also in L because the state is accepting. In both cases, not only is the identified prefix in L, but it is also the *longest* prefix of the remaining input that is in L. If there were a longer prefix of the remaining input also in L, then the DFA would run on the longer prefix without getting stuck, but this contradicts the fact that the DFA did get stuck.

We can use this observation to define the following algorithm to solve the maximal munch scanning problem:

1. Run the DFA for L on the remaining input until the DFA gets stuck or the end of input is reached.
2. If the DFA is in a non-accepting state, backtrack the DFA and the input to the last-visited accepting state. If no accepting state was visited, there is no solution, so output an error.
3. Output a token corresponding to the accepting state that the DFA is in.

4. Reset the DFA to the start state and repeat, starting again from Step 1.

In Step 2, it is sometimes necessary to backtrack the DFA and the input to the last-visited accepting state. An easy way to implement this is to keep a record of the last accepting state that was visited and the position in the input at which it was visited. That way, when the DFA gets stuck in a non-accepting state, we can backtrack just by returning this recorded state and input position. In Assignment 7, you will implement Steps 1 and 2 in the `scanOne` method, whose signature is:

```
322 def scanOne(input: List[Char],
323              state: State,
324              backtrack: (List[Char], State)
325             ): (List[Char], State)
```

The `input` parameter is the remaining input. The `state` parameter is the current state of the DFA. The `backtrack` parameter records the input position and DFA state the last time the DFA visited an accepting state. To signal that the DFA has scanned a token, `scanOne` returns the input remaining after the token and the state of the DFA after processing the token. When the DFA gets stuck in a non-accepting DFA state, `scanOne` can implement the backtracking by simply returning its `backtrack` parameter.