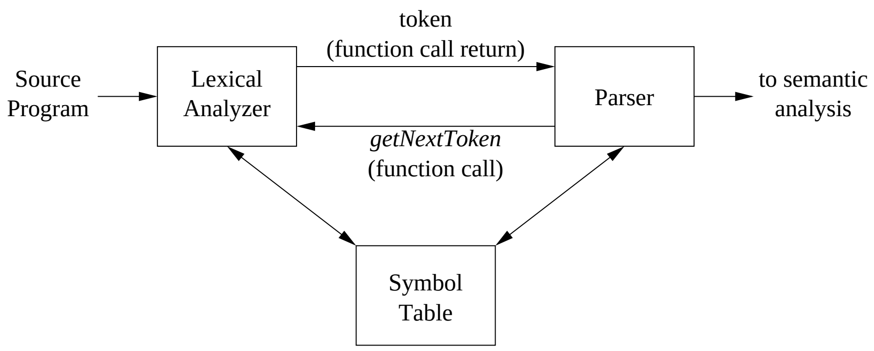
Lexical Analysis

Read Chapter 3 of the Dragon Book (you can skip sections 3.2 and 3.9) or Chapter 2 of the Mogensen book (you can skip sections 2.7, 2.8, and 2.10).

Introduction

So far we have assumed that the lexical analyzer converts a stream of input characters into a stream of tokens that becomes the input to the parser. In practice the lexer is simply a function that is called by the parser every time the parser wants to get the next token. When the lexer is called by the parser, the lexer reads more input characters, matches those input characters using regular expressions until the next lexeme has been found, and then returns the corresponding token to the parser. The value of the lexeme itself can be passed to the parser through the symbol table, if necessary (or through some global variable like `yylval` if the lexer and parser have been generated using Lex and Yacc, which we will look at later in this course):



So in practice the lexer and parser act as a producer-consumer pair, where the lexer produces tokens on demand every time the parser needs to consume more tokens. That way the lexer does not need to read the whole source program at once, but only has to read enough of it to find the next lexeme.

In addition to matching input, the lexer is also usually in charge of removing comments and white spaces from the source program, since these are of no use to the rest of the compiler (some programming languages, like Python or Haskell, rely on spaces to determine the start and end of blocks of statements though, the way C uses { and }, so lexers for Python or Haskell have to produce tokens representing spaces).

The lexer is also in charge of putting identifiers in the symbol table of the compiler. In addition to the identifier itself, the lexer will also put in the symbol table entry the line number in the source program where the identifier appears. That information will be used when an error message needs to be printed for that identifier. The parser usually does not know and does not care how many lines of the source program have been processed so far, only the lexer keeps track of that.

There are several reasons for separating the lexer and the parser.

- Separating the two makes each simpler to implement. Implementing a single phase that combined both lexing and parsing at the same time would make that piece of software very complex. Try to write a context-free grammar for simple arithmetic expressions that takes into account all possible combinations of spaces and you will see that the grammar soon becomes enormous.

- Once you have a separate lexer it is easier to specialize it to do a lot of input buffering (i.e. reading the source program in big blocks of characters instead of character by character) which makes the lexer much faster. Buffering also makes it easy for the lexer to "put back" characters into the input, which makes it easy to implement lookahead tokens, which is necessary for some parsers.

- It becomes easier to change the compiler to support special characters or foreign keywords since these changes are then restricted to the lexer. For example, some old computer keyboards do not have the `[` and `]` characters, so the C language provides two "trigraphs" `??(` and `??)` so people can simulate typing those characters. The lexer will then internally change `??(` into `[` and `??)` into `]`. Foreign keywords (for example, using `si-alors-sinon` instead of `if-then-else`) can be supported simply by changing the corresponding regular expressions in the lexer (more on regular expressions below). In essence the lexer isolates the parser from the way the lexemes in the source code are actually written.

Our study of lexers is going to have two parts. In the first part we are going to formally define tokens, patterns, lexemes, strings, languages, and regular expressions, to ensure that our study of lexers is on solid formal ground. In the second part we are then going to see how to go from the formal definitions to an implementation of a lexer in C.

Tokens, Patterns, Lexemes, Attributes

An *alphabet* is a set of symbols. These are the symbols that are used to write source programs in some programming language. For example, the C programming language uses the ASCII alphabet, while Java use the Unicode alphabet.

A *string* is defined as a *finite* sequence of symbols from the source alphabet (the alphabet for source programs). Note that this concept of string is different from C strings or Java strings, which are sequence of symbols that appear between two double-quotes ("). Here a string is any sequence of symbols that can appear anywhere in a program (which is why its length cannot be infinite). Note that a string of length zero (ε) is fine.

A *token* is then a set of strings over the source alphabet. We will usually give a meaningful name to the set; for example we will use the name "identifier" for the set of all the strings that can be used as an identifier in a programming language, or "integer_literal" for the set of all the strings that represent integer constants.

Some tokens represent sets of finite size while some other tokens represent sets which are infinite in size. For example a token like "if" will represent the set of strings that only contains the string `if`, while the token "identifier" will represent the infinite set of strings that can be used as identifiers in the programming language.

A *pattern* is a mathematical rule that describes the set of strings for a token. We will use regular expressions (see below) to specify a pattern for each token.

A *lexeme* is then a sequence of alphabet symbols (a string) from the source program that is matched by a pattern, and therefore belongs to the set of strings for the corresponding token. This means that a token is the set of all the possible lexemes that are matched by the corresponding pattern (regular expression).

For example, if you remember from the last lecture notes, the statement:

```
index := 2 * count + 17;
```

is made of 24 characters (including spaces) that a lexer will slice into 8 different lexemes that belong to 6 tokens (sets of lexemes) ("identifier" and "integer_literal" each appears twice):

Lexeme	Token
index	identifier
:=	equal_sign
2	integer_literal
*	mult_op
count	identifier
+	plus_op
17	integer_literal
;	semicolon

Here the tokens equal_sign, mult_op, plus_op, and semicolon are all sets with a finite size (a size of one, in fact), while identifier and integer_literal are tokens that corresponds to sets of infinite size.

In this course we will often use the same name for a token and the corresponding lexeme, when the token is a set that contains only a single lexeme. For example, we will often use the token "if" to represent the set of lexemes that only contains the lexeme `if`. You should nevertheless make a clear difference between a lexeme, which is a piece of the source program, and the corresponding token, which is a set of lexemes.

Here is a table with some examples:

Token	Sample Lexemes	Informal description
if	`if`	the "if" keyword
relat_op	`<  <=  ==  >  >=  !=`	relational operators
identifier	`x  f1o2o3`	a letter followed by any number of letters and digits
integer_literal	`-3  0  234523`	any integer constant
string_literal	`"x"  "f1o2o3"  ""  "\""`	a sequence of characters between two double-quotes(")

Later, when we use the output of the lexer as input to the parser, the tokens will form the terminals that appear at the leaves of the parse tree constructed by the parser. When implementing a lexer, you will actually use unique integers to represent each token instead of using names like "identifier". The lexer and the parser will then just have to agree on which integer represents which token (in the case of Lex and Yacc these integers will be defined in the file `y.tab.h`, which will be generated by Yacc based on the `%token` directive in the grammar specification provided by the user).

Token Attributes

If a token corresponds to more than one lexeme then the lexer has to save more information about the token. For example, we know that the token "if" can correspond to only one lexeme: `if`, so not much more information is needed in this case. But a token like "integer_literal" can correspond to many possible lexemes. The parser in general does not need to know the actual lexeme in the source program that corresponds to a given token, but other phases of the compiler will need to know. For example, the code generation phase needs to know the actual integer that corresponds to a given token integer_literal, so that it can generate the right assembly code with the right value. Other information that the compiler might want to have is the position (line number and column number) where the lexeme appears in the source program.

Each piece of extra information associated with a given token is called an *attribute*. So an "identifier" token will have for example one attribute that gives the value of the associated lexeme (the actual identifier as it appears in the source code) and another attribute specifying the position of the lexeme in the source code. In practice all the attributes for a given token are usually stored as entries in the symbol table, and the lexer returns to the parser not just simple tokens but pairs containing both a token and a pointer to the associated symbol table entry.

So for example for the following code:

```
index := 2 * count + 17;
```

the lexer will not return the tokens identifier, equal_sign, integer_literal, mult_op, identifier, plus_op, integer_literal, and semicolon, but instead will return pairs:

<identifier, pointer to symbol table entry for `index`>
<equal_sign, `null`>
<integer_literal, `2`>
<mult_op, `null`>
<identifier, pointer to symbol table entry for `count`>
<plus_op, `null`>
<integer_literal, `17`>
<semicolon, `null`>

Note that, instead of returning pairs like <integer_literal, `17`>, some compilers will put the value `17` in the symbol table too and replace `17` in the pair with a pointer to the corresponding symbol table entry. Which solution is used in practice is specific to each compiler.

Specification of Tokens

As we have said above, we are going to use regular expressions to specify patterns that describe the set of strings that form a token. To define precisely what regular expressions are, we are first going to give below some precise definitions about strings. Using these definitions for strings we will then define precisely what languages are. Finally we will define regular expressions in terms of the languages that they represent.

Strings

As we have seen, an alphabet (usually written as Σ) is a set of symbols. A string over some alphabet Σ is then a finite sequence of symbols from Σ. We will write $|s|$ to represent the length of the string s. The *empty string* is then the string of length zero, which we write as ε: $|\varepsilon| = 0$ by definition.

A *prefix* of a string s is formed by removing zero or more symbols from the end of s. For example, `banana`, `banan`, `bana`, `ban`, `ba`, `b`, and ε are all prefixes of `banana`. From this definition, every string is also a prefix of itself. The empty string ε is a prefix of every string.

Symmetrically, a *suffix* of a string s is formed by removing zero or more symbols from the beginning of s. For example, `banana`, `anana`, `nana`, `ana`, `na`, `a`, and ε are all suffixes of `banana`. Every string is also a suffix of itself. The empty string ε is a suffix of every string.

A *substring* of a string s is formed by removing both a prefix and a suffix from s. For example, `banana`, `bana`, `nana`, `nan`, `b`, and ε are all substrings of `banana` (and there are more). The prefix and suffix which are removed can both be the empty string, so every string is also a substring of itself. The empty string ε is a substring of every string.

A *proper* prefix / suffix / substring of a string s is any non-empty string t which is a prefix / suffix / substring of s and such that s and t are different. For example, `bana` is a proper prefix of `banana` since it is a prefix of `banana` and is different from `banana` itself.

The *concatenation* of two strings s and t is the string obtained by appending the sequence of symbols in t to the sequence of symbols in s, and is written st. Note that $\varepsilon s = s\varepsilon = s$.

The *exponentiation* of a string s is written as s^i and is defined as follows: $s^0 = \varepsilon$, and $s^i = ss^{i-1}$ for $i > 0$. So, for example, $s^1 = ss^0 = s\varepsilon = s$. Similarly, $s^3 = sss$, etc.

Languages

A *language* is a set of strings over some alphabet Σ. In this definition each string in the set is meant to represent a possible program, so a language is the set of all the possible programs. Note that the set of strings can be infinite in size, and it usually is for any interesting programming language, since any interesting programming language allows you to write an infinite number of different programs.

Here are some example of programming languages:

- $\emptyset$: the programming language that contains no program. This means that it is not possible to write a program in this programming language. A compiler for this programming language simply always rejects any input it is given, since such input can never be a program in this programming language.

- $\{\varepsilon\}$: the programming language that contains only the empty program (a program consisting of only the empty string, which corresponds in C or Java to having an empty source file). The empty program is therefore the only program that can correctly be compiled, all other programs must be rejected by a compiler for this programming language. The empty program is (presumably, based on the most obvious semantics) compiled into an executable program that does nothing.

- {`a`, `aa`, `aaa`, `aaaa`}: the programming language that contains only the four possible programs `a`, `aa`, `aaa`, `aaaa`. All other programs must be rejected by a compiler for this programming language, including the empty string. It is not clear what each of the four programs should be compiled into (you can invent your own semantics for them if you want to. . .)

- {`a`, `aa`, `aaa`, `aaaa`, . . . }: the programming language that contains all the programs consisting of one or more characters `a`. Note that this programming language includes an infinite number of possible programs (like C, or Java, and most other programming languages). The empty string, as well as any program that contains anything any other character apart from `a`, must be rejected by a compiler for this programming language. Again, the semantics of such programming language is left for the reader to specify. . .

The *union* $L \cup M$ of two languages L and M is the set of strings s such that s is either in L or in M. For example, if L is the set of letters and M is the set of digits, then $L \cup M$ is the set of letters and digits.

The *concatenation* LM of two languages L and M is the set of strings st where s is a string in L and t is a string in M. For example, if L is the set of letters and D is the set of digits, then LD is the set of strings consisting of a letter followed by a digit.

The *exponentiation* L^i of a language L is the language defined as follows: $L^0 = \{\varepsilon\}$, and $L^i = LL^{i-1}$ for $i > 0$. From this definition, $L^1 = LL^0 = L\{\varepsilon\} = L$. Similarly, $L^3 = LLL$, which is the set of strings stu where s, t, and u are all strings in L (note that this is different from the set of strings sss where s is in L). For example, if L is the set of letters then L^3 is the set of strings consisting of exactly three letters.

The *Kleene closure* (named after Mr. Kleene) L^* of a language L is the language defined as $\bigcup_{i=0}^{\infty} L^i$. In other words, $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup L^4 \cup \ldots$ Since L^* includes L^0, it means L^* always includes ε. For example, if L is {`a`, `b`}, then L^0 is $\{\varepsilon\}$, L^1 is {`a`, `b`}, L^2 is {`aa`, `ab`, `ba`, `bb`}, L^3 is {`aaa`, `aab`, `aba`, `abb`, `baa`, `bab`, `bba`, `bbb`}, etc., and L^* is then the union of all these sets: {ε, `a`, `b`, `aa`, `ab`, `ba`, `bb`, `aaa`, `aab`, `aba`, `abb`, `baa`, `bab`, `bba`, `bbb`, ..., `abbabbaaaaba`, ...}. So L^* is the set of all possible strings of `a`s and `b`s, including the empty string ε. If D is the set of all digits {`0`, ..., `9`}, then D^* is the set of all possible strings of digits, including the empty string.

The *positive closure* L^+ of a language L is the language defined as $\bigcup_{i=1}^{\infty} L^i$. In other words, $L^+ = L^1 \cup L^2 \cup L^3 \cup L^4 \cup \ldots$ Since L^+ does no includes L^0 it means that L^+ includes ε if and only if L itself includes ε. For example, if L is {`a`, `b`}, then L^+ is {`a`, `b`, `aa`, `ab`, `ba`, `bb`, `aaa`, `aab`, `aba`, `abb`, `baa`, `bab`, `bba`, `bbb`, ..., `abbabbaaaaba`, ...}. So L^+ is the set of all possible non-empty strings of `a`s and `b`s. If D is the set of all digits {`0`, ..., `9`}, then D^+ is the set of all possible non-empty strings of digits, which means D^+ is the set of all strings that represent natural numbers.

From the definition of L^* and L^+ we have the obvious relation $L^* = L^+ \cup \{\varepsilon\}$.

Given such definitions, if L is the set of letters and D the set of digits, then $L(L \cup D)^*$ is the set of all strings of letters and digits that start with a letter (i.e. what is often defined to be the set of identifiers).

Regular Expressions

Regular expressions are a *notation* that allows us to easily specify languages. Each regular expression r denotes (is a notation for) a language $L(r)$. A regular expression r is often built by combining simpler regular expressions together. The corresponding language $L(r)$ is then defined by combining the languages corresponding to the simpler regular expressions using a set of rules that we now describe.

Regular expressions over an alphabet Σ are defined as follows:

- ε is the regular expression that denotes the language $\{\varepsilon\}$, i.e. the language that only contains the empty string. Note that we will often use the symbol ε to represent both the regular expression ε and the empty string ε. This is fine, since the empty string ε is the only string that belongs to the language denoted by the regular expression ε. You should not confuse the two though (a regular expression is not a string and a string is not a regular expression).

- If `a` is a symbol in Σ, then `a` is a regular expression denoting the language $\{$`a`$\}$. Again, we will often use the same symbol for a regular expression and the corresponding unique alphabet symbol that belongs to the set denoted by the regular expression.
- If r is a regular expression that denotes $L(r)$ and s is a regular expression that denotes $L(s)$, then:
  - $(r)|(s)$ denotes the language $L(r) \cup L(s)$. In short, the symbol $|$ in a regular expression acts as an "or" operator.
  - $(r)(s)$ denotes the language $L(r)L(s)$. In short, the concatenation of regular expressions corresponds to the concatenation of the denoted languages.
  - $(r)^*$ denotes the language $(L(r))^*$. The $*$ operator in a regular expression is called the *Kleene star* and corresponds to the Kleene closure of the denoted language. In short, the symbol * in a regular expression means "zero or more instances of".
  - (r) denotes the same language $L(r)$ as r does. In short, extra parentheses do not matter in a regular expression.

This definition of regular expression is recursive. It works in fact by induction on the structure of a regular expression: the meaning of a regular expression can be determined by looking at the meaning of the subparts in the regular expression, and then combining the meaning of the different subparts to get the meaning of the whole regular expression. The first two cases in the definition above (for ε and `a`) serve as the base cases for the induction. Note that this definition does not define any meaning for r^i (though it would be easy to define the meaning of this notation based on the definition of language exponentiation) so do not try to use such notation in homework assignments or exams.

To avoid writing too many parentheses we use the following extra rules:

- The * operator has the highest precedence and is left associative.
- Concatenation has the next highest precedence and is left associative.
- The $|$ operator has the lowest precedence and is left associative.

Using these rules and the fact that the regular expressions r and (r) always represent the same language, we can simplify a regular expression like $(a)|((b)^*(c))$ into $a|b^*c$.

Based on such definitions, we then have the following examples. If Σ is the set {a, b} of symbols, then the regular expression a denotes (is a notation for) the language {a}, the regular expression a|b denotes the language {a, b}, and the regular expression (a|b)(a|b) denotes the language {aa, ab, ba, bb}. The regular expression a* denotes the language {ε, a, aa, aaa, ...}. The regular expression (a|b)* denotes the language {ε, a, b, aa, ab, ba, bb, aaa, aab, aba, ...}, i.e. all the strings containing any number of as and bs in any order. This is not the same as a*|b*, which denotes the set {ε, a, b, aa, bb, aaa, bbb, ...}, i.e. all the strings containing either only as or only bs (but not a mix of both). The regular expression ab* denotes the set {a, ab, abb, abbb, ...} while the regular expression (ab)* denotes the set {ε, ab, abab, ababab, abababab, ...} and the regular expression a|b* denotes the set {ε, a, b, bb, bbb, ...}.

Regular expressions have the following algebraic properties:

- $r|s = s|r$: the | operator is commutative.
- $r|(s|t) = (r|s)|t$: the | operator is associative.
- $r(st) = (rs)t$: concatenation is associative.
- $r(s|t) = rs|rt$ and $(s|t)r = sr|tr$: concatenation distributes over the | operator.
- $\varepsilon r = r = r\varepsilon$: the regular expression ε is the identity element for regular expression concatenation (just like the empty string ε is the identity element for string concatenation).
- $r^* = (r|\varepsilon)^*$: relation between ε and the Kleene star.
- $r^{**} = r^*$: the Kleene star is idempotent.

You can prove all these properties by using the inductive definition given above for regular expressions. Try it.

These properties mean that a given language can be described using many different regular expressions. For example, the language {aa, ab, ba, bb} can be described by each of the following regular expressions: (a|b)(a|b), (aε|b)ε(a|b), a(a|b)|b(a|b), aa|ab|ba|bb. This last regular expression in particular shows that any finite language (any language that is a finite set of strings) can be described using a regular expression that simply lists all the possible strings separated by the | operator.

To simplify the definition of regular expressions, it is often convenient to use *regular definitions*, which are regular expressions that are given names. These names can then be used in other regular expressions as if they were symbols. For example, here is a regular expression defining the language of signed rational numbers: (+|-|ε)(0|1|2|3|4|5|6|7|8|9) (0|1|2|3|4|5|6|7|8|9)*(ε|.(0|1|2|3|4|5|6|7|8|9) (0|1|2|3|4|5|6|7|8|9)*)

This regular expression is quite complex though, so it is convenient to use regular definitions to split this big regular expression into several smaller regular expressions which are then easier to define and understand:

$$\begin{array}{rcl}
\text{digit} & \rightarrow & \texttt{0}|\texttt{1}|\texttt{2}|\texttt{3}|\texttt{4}|\texttt{5}|\texttt{6}|\texttt{7}|\texttt{8}|\texttt{9} \\
\text{natural} & \rightarrow & \text{digit digit}^* \\
\text{unsigned\_rational} & \rightarrow & \text{natural}(\varepsilon|\texttt{.}\text{natural}) \\
\text{signed\_rational} & \rightarrow & (\texttt{+}|\texttt{-}|\varepsilon)\text{unsigned\_rational}
\end{array}$$

Note that, unless otherwise stated, spaces are not relevant in regular expressions.

To simplify things even further, we can defined some new notation for some regular expressions:

- $r^+ = rr^*$: the $^+$ operator means "one or more instance of". This operator corresponds to the positive closure of the corresponding language. We then have the following algebraic property: $r^* = r^+|\varepsilon$.
- $r? = r|\varepsilon$: the ? operator means "zero or one instance of".
- $\texttt{0}|\texttt{1}|\texttt{2}|\texttt{3}|\texttt{4}|\texttt{5}|\texttt{6}|\texttt{7}|\texttt{8}|\texttt{9} = [\texttt{0}-\texttt{9}]$: classes of characters (like digits, or letters) can be abbreviated using an interval notation.

Using these notations, the definition for signed rational numbers above becomes:

$$\begin{array}{rcl}
\text{digit} & \rightarrow & [\texttt{0}-\texttt{9}] \\
\text{natural} & \rightarrow & \text{digit}^+ \\
\text{unsigned\_rational} & \rightarrow & \text{natural}(\texttt{.}\text{natural})? \\
\text{signed\_rational} & \rightarrow & (\texttt{+}|\texttt{-})?\text{unsigned\_rational}
\end{array}$$

These new notations, as well as the use of regular definitions, do not increase the power of regular expressions, but just make it more convenient to write complex regular expressions.

Limitations of Regular Expressions

While regular expressions are powerful enough to describe many simple patterns one might think of, there are many languages that cannot be described using regular expressions. This is the case every time, for example, that describing a language might require counting symbols in one way or another.

For example, regular expressions cannot be used to describe the language of balanced (matched) sequences of parentheses, because defining such a language requires at a minimum to use a notation that can somehow count that there are as many opening parentheses as there are closing parentheses. Another example is the language $\{\texttt{a}^n\texttt{b}^n,\ n > 0\}$ (the language of strings that start with `a`s and end with the same number of `b`s). Writing a regular expression to describe such a language is not possible, since a regular expression cannot check whether two different parts of a string have the same length (which, again, would require counting). A regular expression cannot even check that two different parts of a string are the same, so regular expressions cannot be used to describe a language like $\{\texttt{wcw},\ \texttt{w} \text{ in } (\texttt{a}|\texttt{b})^*\}$ (the language of strings that have the same start and end, with the letter `c` in between).

In general regular expressions can only be used to described a fixed, known, number of repetitions of a known pattern, or an unspecified number of repetitions (using the Kleene star) of a known pattern in the input program.

Tokens and Regular Expressions

As we have just seen, we can use a regular expression to describe the set of strings that belong to a language. If you look at the definitions we have previously given for both a token and a language, you will see that both are defined as a set of strings of symbols over some alphabet. In fact a token *is* a little language. For example a token like integer_literal represents the set of all the strings which represent integer constants in a program, which is a little language of its own. Since we can use a regular expression to describe a language, we can therefore also use a regular expression to describe a token. In fact this is exactly how you are going to define the set of lexemes that form a token when you are going to implement a lexer: by creating a regular expression that describes that set of lexemes. In essence the regular expression will define a pattern and every input string that is matched by the pattern will be part of the token corresponding to the regular expression.

For example, if we want to define the token id of identifiers (variable names, function names, class names, etc.) to be the set of all strings of letters and digits that start with a letter, then we can use regular expressions (and regular definitions) to define the id token as follows:

$$\begin{array}{rcl} \text{digit} & \rightarrow & [\texttt{0}-\texttt{9}] \\ \text{letter} & \rightarrow & [\texttt{a}-\texttt{zA}-\texttt{Z}] \\ \text{id} & \rightarrow & \text{letter}(\text{letter}|\text{digit})^* \end{array}$$

This is very nice, but we still have a big problem: given a regular expression that describes some token, how do we check whether some input characters from a source program are actually matched by the regular expression or not? We need some algorithm to do this, to decide whether the input characters together form a lexeme that belongs to that token. In fact we need to be able to check each piece of input from the source program against many different regular expressions, one for each possible token in the programming language our compiler implements, until we find the token to which the piece of input belongs. This is what we are going to study now.

From Regular Expressions to Lexical Analyzers

Now that we know how to use regular expressions to specify tokens, we have to understand how a sequence of input characters coming from a source program can be matched against a regular expression and sliced into lexemes. So we have to understand how a given regular expression can be transformed into the code of a lexer that, once compiled into an executable lexer, will try to match input characters to check whether they belong to the token (set of strings) described by the given regular expression.

The process to go from a regular expression to the code of a lexer is going to involve three steps:

- first the regular expression (RE) will be transformed into a *nondeterministic finite automaton* (NFA) that can recognize input strings that are matched by the RE;
- since computers are deterministic machines, we will then have to transform the NFA into a *deterministic finite automaton* (DFA);
- finally we will see how to implement a DFA using simple C code, which will then be the C code of a lexer that can recognize input strings that are matched by the original RE.

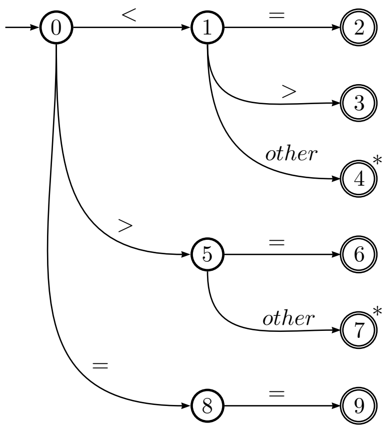
There are other methods (that go directly from regular expression to DFA, for example) but the one we are going to use here is overall the simplest one, even though it has three separate steps.

Before describing these three steps, we first explain how DFAs can be used to match input characters and therefore why we want to transform regular expressions into C code that implements such DFAs.

Using Deterministic Finite Automata to Match Input

Consider for example the problem of identifying relational operators in the input of the lexer. The token "relat_op" is a set of six possible lexemes: `<=`, `<`, `==`, `>`, `>=`, and `<>` (using a Pascal-like "not equal" operator). Since this set is finite, one possible regular expression that describes this set of strings is the regular expression that simply enumerates the lexemes: `<=|<|==|>|>=|<>`. An equivalent regular expression is `<(=|>|`ε`)|>(=|`ε`)|==`, where the `<` and `>` characters have been factored out.

This second regular expression corresponds to the following DFA (based on Figure 3.12 in the Dragon Book):



The DFA starts in its start state, which is state 0. When the string `<=` is given as input to this DFA, for example, the DFA reads the first input symbol `<` and therefore moves from state 0 to state 1 through the transition labeled `<`. Then the DFA reads the second input symbol `=` from the input and moves from state 1 to state 2. Since state 2 is a final state (as indicated by the fact that it is circled twice), the DFA has therefore identified the string `<=` in the input, which is thus a lexeme, and the lexer that implements the DFA can then return the corresponding token relat_op to the parser. Of course some later phases of the compiler (like the code generation phase) will need to know exactly which relational operator was seen in the input, so what the lexer actually returns to the parser is a pair <relat_op, LE>, where LE is a symbolic constant that represents the "less than or equal" operator. The relat_op token in the pair is used by the parser to construct a parse tree, and the associated symbolic constant is used by other phases in the compiler that need to know the exact relational operator. Alternatively, the lexer can put the lexeme `<=` into the symbol table and return as a pair the token relat_op and a pointer to the corresponding symbol table entry.

Similarly, if the input given to the DFA is the string `<>`, then the DFA goes from state 0 to state 1 to state 3, and then returns to the parser a pair <relat_op, NE>, where NE is a symbolic constant representing the "not equal" operator (or, alternatively, use the symbol table and return a pointer to the corresponding entry).

If the input given to the DFA is the string `<3`, then the DFA moves from state 0 to state 1 on reading the first symbol `<`. When the DFA reads the second input symbol `3`, it moves from state 1 to state 4 since the label on the transition from state 1 to state 4 is "other". The label "other" means that this transition should be taken every time the input symbol being read is anything other than `=` or `>` (the labels of the other two transitions out of state 1). When the DFA reaches state 4 the lexer has therefore identified the string `<` (followed by some other symbol) and can return a pair like <relat_op, LT> to the parser, where LT is a symbolic constant representing the "strictly less than" operator. Just before the lexer does this, however, it has to put back into the input the symbol `3` that was read by the DFA to move from state 1 to state 4. That symbol is not part of the lexeme `<` and the DFA needed to read that extra symbol only to be able to make the difference between `<`, `<=`, and `<>`: if the DFA had not read that extra symbol then it would not have been able to ensure that the lexeme is not `<=` or `<>`. Since the extra symbol that was read is not part of the lexeme, it has to be put back into the input so that it can be processed again later by the lexer the next time the parser calls the lexer. This is what the "$*$" next to state 4 represents: the last input symbol that was read to go through the "other" transition has to be put back into the input. So if the input was for example `<3`, then the symbol `3` has to be put back into the input to be processed again later by the lexer (and, presumably, then be identified as a lexeme for an integer).

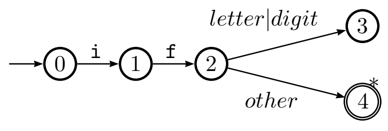
The rest of the DFA works similarly as the previous cases above. The only thing left to explain is what happens when the DFA receives the input `=3`. In that case the DFA moves from state 0 to state 8. Since the next input symbol after that is not a second `=` symbol but the symbol `3`, the DFA gets stuck, meaning that it cannot move to any other state. Since state 8 is not a final state of the DFA, the DFA then rejects the input, which proves that the string `=` is not a relational operator. The lexer then will presumably try other DFAs on the original input `=3`, until it finds the one that matches the string `=`.

Note the difference between this case and the previous one: here the DFA did not detect a lexeme in the input that matches the regular expression implemented by the DFA, so the DFA rejects the whole input and the lexer has then to try other DFAs for that input (if all DFAs reject that input, then the input is not part of the programming language and the lexer has then detected a lexical error). In the previous case a lexeme was found in the input, a symbol that was read but was not part of the lexeme had to be put back into the input, and a token was then returned to the parser.

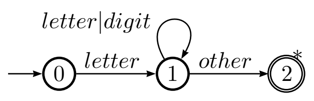
Note also that the finite automaton above is deterministic since no state has two different transitions out of the state labeled with the same symbol. This means that, in a given state, there is only at most one possible choice for making a transition on a given input symbol. In other words, the automaton always knows what to do next just by looking at its current state and the next symbol in the input. If that were not the case, then the automaton would have to make random choices, and maybe later backtrack when a choice proved to be the wrong one. Using a DFA instead of an NFA makes matching input much simpler and more efficient.

Here is another example of matching input using DFAs. The goal of this example is to recognize identifiers (a letter followed by a mix of letters and digits) as well as reserved keywords (like `if`, `for`, `while`, etc.) The difficulty here is that reserved keywords look very much like ordinary identifiers. For example the strings `i` and `ife` both are ordinary identifiers in the C or Java programming languages, but the string `if` is a reserved keyword.

The first solution to this problem is to use a separate specialized DFA for each keyword plus one more general DFA for identifiers, and try each of them one by one on the input, in the right order. For example here is a DFA recognizing the reserved keyword `if` (but not the two letters `if` when they are a prefix of a longer identifier like `ife`):



and the more general DFA recognizing all identifiers (see Figure 3.13 in the Dragon Book):



This second DFA accepts any string that starts with a letter followed by any mix of letters or digits. When the DFA reads something other than a letter or digit (a space character, for example) it moves from state 1 to state 2, puts back into the input the last symbol read (as indicated by the "$*$") since that last symbol read is not a letter or digit and therefore is not part of the lexeme, and then returns the token "identifier" to the parser.

This second DFA will obviously match both identifiers like `i` and `ife` as well as reserved keywords like `if`. To ensure that the string `if` is correctly recognized as a keyword and not as an identifier, the lexer has therefore to ensure that the first DFA is always tried on the input before the second one is tried. In general there will be a separate DFA for each reserved keyword like `if`, `for`, `while`, etc., and the lexer will have to try each of these DFAs first, one by one, to see whether the input corresponds to a reserved keyword or not. If none of these DFAs matches the input, the lexer will then and only then try the more general DFA for identifiers. This ensures that keywords are always recognized as keywords, and that identifiers are always recognized as identifiers.

This method of having a separate specialized DFA for each reserved keyword works fine, but it makes the lexer quite complex, since the lexer needs then to manipulate many DFAs. A programming language like C or Java has dozens of reserved keywords, which means the lexer has to contain dozens of specialized DFAs just to properly identify these reserved keywords.

A simpler method to recognize both reserved keywords and identifiers is to use only the second more general DFA above, along with extra information in the compiler's symbol table to differentiate between what is a reserved keyword and what is an identifier.

To do this, the compiler first puts in the symbol table all the reserved keywords and marks them as such in the table. So before the lexer is even called the symbol table will already contain many entries, one for each reserved keyword in the programming language. When the lexer runs, it then uses only the second general DFA above. This DFA will recognize both reserved keywords and identifiers as being identifiers. E.g. the DFA will not make any difference between identifiers like `i` and `ife` and reserved keywords like `if`. When some input has been matched by the DFA, and just before the lexer returns a token to the parser, the lexer will use the lexeme that has just been matched by the DFA and look up that lexeme in the symbol table. Three actions are then possible:

- If the lexeme is not already in the symbol table then the lexer knows this lexeme is an identifier. The lexer then puts the lexeme in the symbol table (all identifiers need to have an entry in the symbol table, because different phases of the compiler need such an entry to store extra information about the identifier, like its type, etc.), marks this lexeme in the symbol table as an identifier, and returns an "identifier" token to the parser (along with a pointer to the new symbol table entry).

- If the lexeme is already in the symbol table and is marked as a reserved keyword, then the lexer simply returns to the parser the corresponding token for that reserved keyword.

- If the lexeme is already in the symbol table and is marked as an identifier, then the lexer is just seeing an identifier it has already seen before, in which case it just returns the "identifier" token to the parser (along with a pointer to the existing symbol table entry).

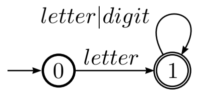
By using the extra information stored in the symbol table the lexer can therefore use a single DFA to correctly recognize all identifiers and all reserved keywords.

This method of using a single DFA to match both identifiers and reserved keywords makes the implementation of the lexer much simpler. It also makes it very easy to change the reserved keywords of a programming language. For example, if one wants to change the programming language to use French reserved keywords instead of English ones, one only has to change the way the symbol table of the compiler is initialized to replace the entry for `if` in the symbol table with an entry for `si`, etc.

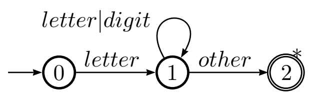
Using DFAs like this, the lexer can identify all lexemes and decide which token (set of lexemes) they belong to. The lexer simply tries every DFA in turn, one by one. If a DFA does not match the input then the lexer tries the next DFA on the input, until it finds one that matches the input. If a DFA matches the input, the lexer then returns the corresponding token to the parser. Later the parser will call the lexer again and the lexer will again use the DFAs to try to match more of the input. If no DFA matches some input then the lexer knows that that input is not a correct lexeme for the programming language. At that point the lexer will reject the input and print an error message indicating that a *lexical error* has been detected.

Now that we know how lexers use DFAs, we have to see how to transform the regular expressions that are used to specify tokens into C code that implements the corresponding DFAs. As indicated above, we are going to do this by going from regular expression to NFA to DFA to C code. Note that in what we are going to see below we will no longer talk about "other" transitions. We will see the various transformations from regular expression to NFA to DFA to C code as mathematical transformations, separately from the problem of lexical analysis. In practice, when a lexer uses DFAs it will slightly modify the DFAs to insert the kind of special "other" transitions that we have seen above. This modification is not difficult to do: if a final state of a DFA has transitions out of it, then the lexer knows it needs to add such "other" transition from this state to a new final state. That new "other" transition is necessary so that the lexer can decide whether it has reached the end of a lexeme in the input or not.

For example, starting from the regular expression letter(letter|digit)* the transformations presented below will compute the following DFA:

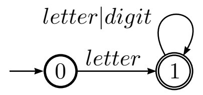


Since this DFA has a transition out of its final state (to the final state itself, in the present case) the lexer will slightly transform this DFA before it can use it to match lexemes, by adding an "other" transition, to get the general DFA for identifiers that we have seen before:



This small transformation is necessary so that the lexer can decide whether it needs to read more input to match more letters or digits, or whether it has reached the end of the lexeme in the input (the lexer knows it has reached the end of the lexeme precisely when it sees in the input something which is not a letter or digit, which is represented by the transition on “other”).

As we just said though, we will not worry about “other” transitions in the explanations below, and instead concentrate on the hard part, which is how to transform a regular expression like letter(letter|digit)* into a DFA like:



and then into the corresponding C code. We will talk again about “other” transitions only once we know how to generate that C code.

From Regular Expression to Nondeterministic Finite Automaton

Before we explain how to transform a RE into an NFA, we first define what NFAs are and how they work.

Nondeterministic Finite Automaton

A *recognizer* for a language L is a program that takes as input a string s and responds “yes” if $s \in L$ and responds “no” otherwise. Finite automata are transitions diagrams (diagrams with transitions between states) that can act as recognizers.

A *nondeterministic finite automaton* is a 5-tuple $<S, \Sigma, move, s_0, F>$, where:

- S is the set of states in the NFA;
- Σ is the alphabet for the input string given to the NFA;
- $move(s, c)$ is the *transition function* that specifies which states of S the automaton can move to when seeing input symbol c while in state s ($move(s, c)$ is therefore always a subset of S);
- s_0 is the *start state* of the NFA;
- F is the set of *final states* (also called *accepting states*) of the NFA.

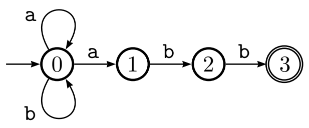
We will often represent NFAs as diagrams with one node for each state in S, and one edge between nodes for each transition between states. The edges will be labeled using the input symbol on which the transition occurs. The start state will have an unlabeled arrow into it and the final states will be circled twice.

An NFA then *accepts* a string t if and only if there is a path in the NFA that starts at s_0 and ends at a final state, such that the labels of the edges in the path together exactly spell the string t. Otherwise the NFA *rejects* the string t. The language defined by an NFA is then the set of strings the NFA accepts.

There are two important things to note about the definition of an NFA:

- *move*(s, c) is not a single state but a set of states, which means that the NFA can in general move to any state among many different possible states when seeing symbol c in the input while in state s. This is what makes the NFA nondeterministic: several transitions to different states are possible and the NFA randomly chooses one among those.
- remember that, for any string s, $\varepsilon s = s\varepsilon = s$. It is therefore possible to insert as many ε as we want anywhere in any input string. These ε can be used by the *move* transition function to make the NFA move from one state to another. Such transitions on seeing an ε in the input are called *epsilon transitions*.

For example, here is an NFA that recognizes the language $(\texttt{a}|\texttt{b})^*\texttt{abb}$, given the alphabet {`a`, `b`}:



The *transition table* for an NFA is a table describing the *move* transition function, with one row for each of the states of the NFA and one column for each of the symbols of the input alphabet.

For example the transition table for the NFA just above is:

	a	b
0	{0,1}	{0}
1	∅	{2}
2	∅	{3}
3	∅	∅

When the NFA is in state 0 and reads input symbol `a`, the table shows that it can move to two different states: 0 or 1. This clearly shows that the NFA really is nondeterministic.

When this NFA is given the input string `abb`, the NFA starts in its start state 0 and reads input symbols one by one. When reading the first input symbol `a` the NFA has two choices: move through the loop from state 0 to state 0 that is labeled with the same symbol `a`, or move from state 0 to state 1 by following the edge labeled with the same symbol `a`. Since the NFA is nondeterministic the NFA can therefore be in two different possible states after reading the input characters `a`: state 0 or state 1.

The NFA then reads the next input symbol which is `b`. If the NFA is in state 0 then it can move from state 0 to state 0 through the loop labeled with `b`. If the NFA is in state 1 then it can move from state 1 to state 2 through the edge labeled with `b`. After reading the input `ab` the NFA can therefore be in two different possible states: state 0 or state 2.

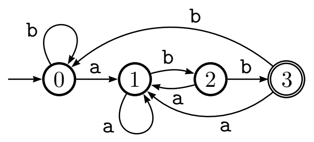
The NFA then reads the next input symbol which is another `b`. If the NFA is in state 0 then it can move again from state 0 to state 0 through the loop labeled with `b`. If the NFA is in state 2 then it can move from state 2 to state 3 through the edge labeled with `b`. After reading the input `abb` the NFA can therefore be in two different possible states: state 0 or state 3.

At that point the NFA has read all the input. Since one of the possible states in which the NFA can be at this point (state 0 or state 3) is a final state (state 3) the NFA therefore accepts the input string. This is what we expect since the string `abb` is in the set of strings represented by the regular expression $(\texttt{a}|\texttt{b})^*\texttt{abb}$.

If the NFA is given the input `ab` then, as above, the NFA will be in two different possible states after reading the whole input `ab`: either state 0 or state 2. At that point the NFA will have read all the input but will not have reached a final state. The NFA will therefore reject the input string `ab`. This is what we expect since the string `ab` is not in the set of strings represented by the regular expression $(\texttt{a}|\texttt{b})^*\texttt{abb}$.

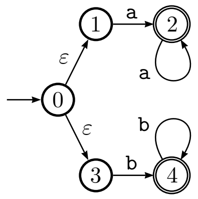
If the NFA is given the input `abbb` then, as above, the NFA will be in two different possible states after reading the input `abb`: state 0 or state 3. At that point the NFA still has to read one more input symbol `b`. If the NFA is in state 0 then it can move from state 0 to state 0 through the loop labeled with `b`. If the NFA is in state 3 then there is nothing it can do: there is no transition on `b` (or any other symbol, for that matter) out of state 3. At that point the NFA is stuck: it needs to consume the input symbol `b` but no transition out of state 3 allows this. So after reading the whole input `abbb` the only state the NFA can possibly be in is state 0. Since state 0 is not a final state the NFA therefore rejects the input. This is what we expect since the string `abbb` is not in the set of strings represented by the regular expression $(\texttt{a}|\texttt{b})^*\texttt{abb}$.

Note that the NFA above is quite simple to understand when looking at it, but using it is complicated because we have to keep track of all the possible states in which the NFA can be at any point in time. What we really want is to have an equivalent DFA for which we will only have to keep track of a single state at any point in time. Here is such a DFA which is equivalent to the NFA above for $(\texttt{a}|\texttt{b})^*\texttt{abb}$:



You can check that this DFA behaves like the NFA above when given various input strings. It is not clear though how to create this DFA from the regular expression (a|b)*abb. This is why, instead of trying to go directly from a regular expression to an equivalent DFA, we will instead use below a simpler method that first constructs an NFA from a regular expression, then transforms that NFA into an equivalent DFA. More on this below.

Here is another example of an NFA:



This NFA implements the regular expression aa*|bb*.

To write the transition table for this NFA we have to add a new column for the ε-transitions:

	a	b	ε
0	$\emptyset$	$\emptyset$	{1,3}
1	{2}	$\emptyset$	$\emptyset$
2	{2}	$\emptyset$	$\emptyset$
3	$\emptyset$	{4}	$\emptyset$
4	$\emptyset$	{4}	$\emptyset$

When this NFA is given the input string aa, it starts in the start state 0. The NFA then may (or may not) immediately move to state 1 or to state 3 through the ε-transitions without consuming any input. Before reading any input the NFA can therefore be in three different possible states: state 0, state 1, or state 3.

The NFA then reads the first input symbol a. If the NFA was in state 0 then it becomes stuck since there is no transition out of state 0 labeled with a. Similarly if the NFA was in state 3 then it becomes stuck since there is no transition out of state 3 labeled with a. If the NFA was in state 1 then it can move to state 2 through the transition labeled with a. After reading the input a the NFA can therefore be only in one possible state: state 2.

The NFA then reads the second input symbol a. The only possible move from state 2 on input symbol a is to go through the loop labeled with a. After reading the input aa the NFA can therefore be only in one possible state: state 2.

At this point the NFA has read all the input and is in a final state (state 2). The NFA therefore accepts the input string aa. This is what we expect since the string aa is in the set of strings represented by the regular expression aa*|bb*.

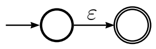
If the NFA is given the input aab then, as above, the NFA will only possibly be in state 2 after reading the input aa. At that point the NFA still has to read the next input symbol b. State 2 has no possible transition out of it on symbol

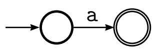
b. The NFA is therefore stuck in state 2 and cannot move to actually consume the last symbol b in the input. The NFA is then in a final state (state 2) but it has not consumed the whole input string. The NFA therefore has to reject the whole input string. This is what we expect since the string aab is not in the set of strings represented by the regular expression aa*|bb*.

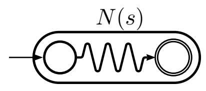
Thompson's Construction Method

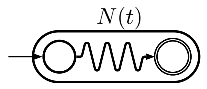
Now that we know exactly what NFAs are and how they work, we can see how to construct NFAs starting from regular expressions. The method we are going to use here is called *Thompson's construction method*. The idea is to split a regular expression into small pieces, then construct an NFA for each small piece, then combine the small NFAs together into a bigger NFA that implements the whole regular expression. In essence the construction of the NFAs is going to be bottom-up: we will first build NFAs to recognize ε and the various input symbols, then expand those NFAs by combining them using various fixed patterns for concatenation, alternation ("or"), and the Kleene star. So we are going to build NFAs in very much the same way as we defined regular expressions above.

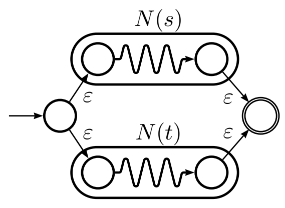
Here is how it works. For a given regular expression r we write $N(r)$ for the NFA that accepts the language $L(r)$. We construct $N(r)$ based on the following rules, which look at the shape of the regular expression r:

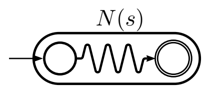
- If $r = \varepsilon$ then $N(r)$ is: 

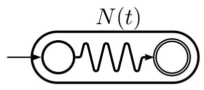
- If $r =$ a, with a $\in \Sigma$, then $N(r)$ is: 

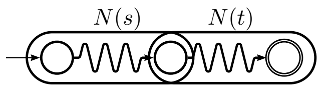
- If $r = s|t$ then:
  - Construct $N(s)$: 

  - Construct $N(t)$: 

  - Combine $N(s)$ and $N(t)$ to get $N(r)$: 

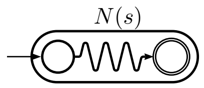
  Note that, after $N(s)$ and $N(t)$ have been combined, these two NFAs do not themselves have start and final states in the resulting NFA anymore.
- If $r = st$ then:
  - Construct $N(s)$: 

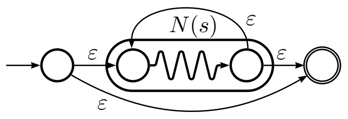
  - Construct $N(t)$: 

  - Combine $N(s)$ and $N(t)$ to get $N(r)$: 

  Note that, after $N(s)$ and $N(t)$ have been combined, these two NFAs share a state: the final state of $N(s)$ and the start state of $N(t)$ have been combined into a single normal state. The start state of $N(s)$ is now the start state of $N(r)$ and the final state of $N(t)$ is now the final state of $N(r)$.

- If $r = s^*$ then:

  - Construct $N(s)$: 

  - Then $N(r)$ is: 

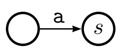
  The bottom ε-transition allows $N(r)$ to go directly from the start state to the final state without going through $N(s)$. This corresponds to going through $N(s)$ zero times. $N(s)$ by itself implements going through $N(s)$ exactly once. The ε-transition going from right to left then allows the NFA to go back to the start of $N(s)$. This corresponds to going through $N(s)$ more than once. Together the possibilities of going through $N(s)$ zero times, one time, or more than one time, correspond to going through $N(s)$ zero or more times, which is what the Kleene star represents. Note that $N(s)$ does not itself have a start and final state in the resulting NFA anymore.

- If $r = (s)$ then $N(r)$ is the same as $N(s)$.

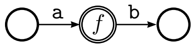
Thompson's construction method has several interesting properties:

- Every NFA has exactly one start state and one final state. If you use Thompson's construction method and end up with an NFA which has more than one start state or more than one final state then you know you made a mistake somewhere.

- Every state of an NFA always has at least one transition going out of it, except for the final state which has zero transition going out of it. In other words, an NFA constructed using Thompson's method cannot contain a state s like this:

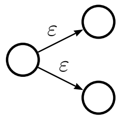


with no transition going out of it, and cannot contain a final state f like this:

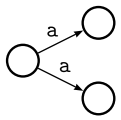


with a transition going out of it. If you use Thompson's construction method and end up with an NFA which has no transition going out of a normal state, or which has one or more transitions going out of the final state, then you know you made a mistake somewhere.

- An NFA constructed using Thompson's method never contains any state with multiple outgoing transitions on the same input symbol. States with multiple outgoing ε-transitions are possible though. In other words, an NFA constructed using Thompson's method can contain states like this:



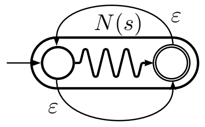
but never states like this:



Again this is something you might want to check when using Thompson's construction method, to help you detect any error you might have made.

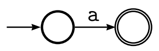
- Each step of the construction introduces at most two new states: one new start state and one new final state (in fact the construction rule for concatenation actually decreases the total number of states by one). This means that the resulting NFA has a number of states which is at most twice the number of symbols and operators in the original regular expression. For example, the regular expression (`a`|`b`)*`abb` contains five alphabet symbols (`a` or `b`) and two operators (| and *, we do not count concatenation as an operator here since regular expressions do not explicitly contain a separate operator for concatenation). This means that, after constructing the corresponding NFA using Thompson's method, this NFA will have at most $(5 + 2) * 2 = 14$ states (in fact it will have 11 states, as we will see below). This is important because in the end we want lexers to use automata which are as small as possible, to make lexical analysis as fast as possible.

Note that the rules presented above are not the only possible rules that can be used to construct an NFA from a regular expression. For example, here is another way to construct $N(s^*)$ given $N(s)$:

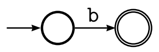


There are several other methods to construct an NFA for s^*, as well as other methods to construct the NFAs corresponding to "|" or to concatenation (see for example the Mogensen book). In these lecture notes we will always use the same construction method as indicated by the rules above though, and these are also the rules you will have to use in homework assignments and exams.

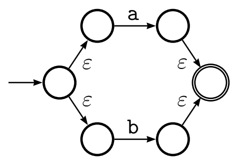
Let's look at two different examples to see how to use the rules above to construct NFAs from regular expression. In the first example we want to construct the NFA corresponding to the regular expression (`a`|`b`)*`abb`. To do this we start by constructing the NFA $N($`a`$)$ for the simple regular expression `a` using the rule for alphabet symbols above:



and the NFA $N($`b`$)$ for the simple regular expression `b` using the same rule:

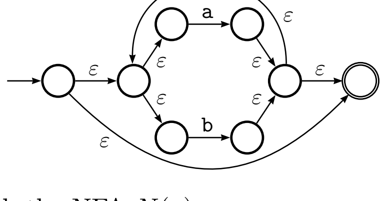


Then we combine these two NFAs together using the rule for "|" to get $N($`a`|`b`$)$:

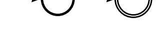


Note that, as we do this, and as indicated by the rule for "|" above, the two NFAs corresponding to $N($`a`$)$ and $N($`b`$)$ do no longer have start and final states in the NFA for $N($`a`|`b`$)$.

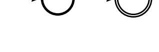
Once we have constructed $N($`a`|`b`$)$, the last rule above says that $N(($`a`|`b`$))$ is the same NFA. We can then construct the NFA $N(($`a`|`b`$)^*)$ using the construction rule for the Kleene star:



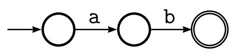
Starting again with the NFA $N($`a`$)$:



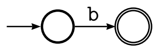
and the NFA $N($`b`$)$:



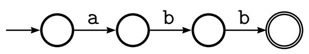
we combine them using the rule for concatenation to get the NFA $N($ab$)$:



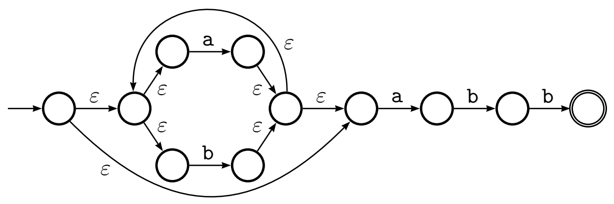
We then use another NFA $N($b$)$:



and again use the concatenation rule to combine it with the previous NFA to get $N($abb$)$:

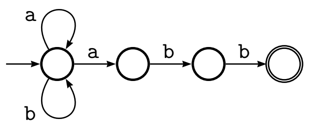


Finally we concatenate the NFA $N(($a|b$)^*)$ with the NFA $N($abb$)$ to get the NFA $N(($a|b$)^*$abb$)$:



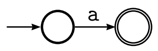
Note that the order in which we concatenate the different parts does not matter. We can construct and concatenate the different parts of the NFA in any order we want and we will still end with the same result. Of course, regardless of which order we use, we will still have to construct $N(($a|b$))$ before we construct $N(($a|b$)^*)$.

Note also that this NFA is quite different and has many more states than the other NFA we have seen at the beginning of these lecture notes for the same regular expression $($a|b$)^*$abb:

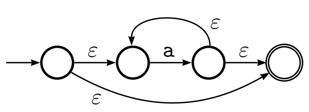


So Thompson's construction method obviously does not construct the smallest possible NFA.

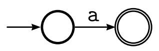
As another example of how to use Thompson's construction method, let's construct the NFA for aa$^*|$bb*. We start again by constructing the NFA $N($a$)$:



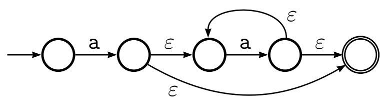
and then use it to construct the NFA $N($a$^*)$ using the construction rule for the Kleene star:



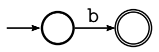
We then construct an NFA $N($a$)$ again:



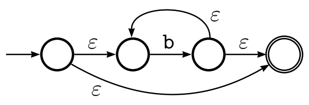
and concatenate it with the previous NFA to get the NFA $N(\texttt{aa}^*)$:



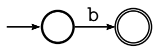
Similarly, we construct the NFA $N(\texttt{b})$:



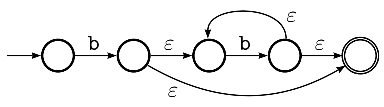
and then use it to construct the NFA $N(\texttt{b}^*)$ using the construction rule for the Kleene star:



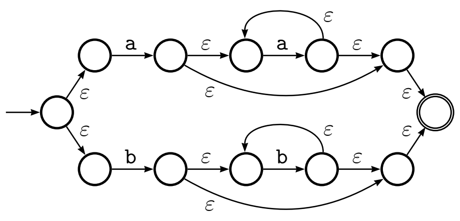
We then construct an NFA $N(\texttt{b})$ again:



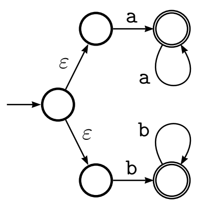
and concatenate it with the previous NFA to get the NFA $N(\texttt{bb}^*)$:



Once we have constructed the NFAs $N(\texttt{aa}^*)$ and $N(\texttt{bb}^*)$, we can then use the construction rule for "|" to construct the NFA $N(\texttt{aa}^*|\texttt{bb}^*)$:



Again, this NFA is quite different from the NFA we have seen before for the regular expression $\texttt{aa}^*|\texttt{bb}^*$:



From Nondeterministic Finite Automaton to Deterministic Finite Automaton

Deterministic Finite Automaton

Before we look at how to transform an NFA into a DFA, let's first formally define what a DFA is. We have seen before that an NFA is a 5-tuple $<S$, Σ, *move*, s_0, $F>$. A *deterministic finite automaton* is simply a special case of NFA where:

- there are no ε-transitions;
- no state has more than one outgoing transition for the same alphabet symbol.

This definition implies that the transition table for the *move* transition function does not have a column for ε-transitions, and that each entry in the table contains at most a single state (in other words, each set $move(s, c)$ is either the empty set or a singleton set).

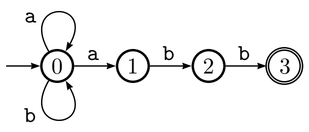
This definition also obviously implies that every DFA is an NFA. So a DFA accepts or rejects input in the exact same way as an NFA does.

The Subset Construction Algorithm

Now that we know how to transform a regular expression into a nondeterministic finite automaton, we have to look at the second step in the process that will take us from regular expression to C code: transforming the nondeterministic finite automaton we have just created using Thompson's construction method into a deterministic finite automaton.

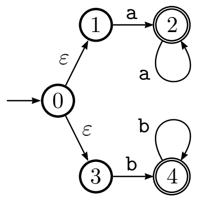
Note that we could try to implement the NFA directly into C code without going through the DFA stage first. But then, when using that C code in a lexer to try to match some input, the program would end up having to either make nondeterministic choices and then later maybe backtrack if at some point the C code made the wrong nondeterministic choice, or the program would end up having to keep track of all the possible states the NFA might be in at every point in time and follow at every step all the possible transitions the NFA might make (including ε-transitions). Implementing backtracking in a programming language like C is not pleasant at all (it is much easier to implement in functional or logic programming languages though). Tracking multiple NFA states at the same time is possible (some tools that use regular expressions, like some editors use for searching text, do that instead of computing a DFA first) but it then makes the lexer slower compared to using a DFA which only has to keep track of a single state at any point in time (interactive editors might not care much about searching fast, but lexers care about matching input fast). So, rather than try to directly implement an NFA in C, what we are going to do instead is to transform the NFA into a DFA, which we will then implement in C. Since, by definition, the DFA will be deterministic, implementing it in C will then be straightforward.

To understand how to transform an NFA into a DFA, let's look at a few examples. If you remember from the section above about nondeterministic finite automata, we observed that when the following NFA:



is in state 0 and is given the input `a`, it has two choices: going through the loop from state 0 to state 0 that is labeled with the same symbol `a`, or move from state 0 to state 1 by following the edge labeled with the same symbol `a`. Since the NFA is nondeterministic the NFA can therefore be in two different possible states after reading the input characters `a`: state 0 or state 1. If we want to simulate this behavior using a DFA, then this DFA needs to have a way to keep track of the fact that the corresponding NFA can be in two possible states at the same time. This means that a single state of the DFA will have to simulate multiple possible states of the NFA!

Let's look at another example. We have also observed before that when the following NFA:



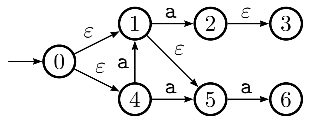
is in the start state 0, it may (or may not) immediately move to state 1 or to state 3 through the ε-transitions without consuming any input. Before reading any input that NFA can therefore be in three different possible states: state 0, state 1, or state 3. Again, if we want to simulate this behavior using a DFA, this DFA then needs to have a way to keep track of the fact that the corresponding NFA can be in three possible states right from the start. This means that again a single state of the DFA (the start state of the DFA, in the present case) will have to simulate multiple possible states of the NFA! This also means that, when constructing the DFA corresponding to the NFA, we will have to keep track of all the possible ε-transitions the NFA might take at each step.

In general, the DFA will have to track at every step the *set* of states the corresponding NFA might possibly be in at that point in time. This means that each DFA state will correspond to a set of NFA states. The DFA will then in essence simulate at every step all the possible moves that the corresponding NFA could make during the corresponding step, based on the set of possible states the NFA might be in at that point.

Before we look at the actual algorithm to compute a DFA starting from an NFA, we must first define three functions that we will need.

- ε-closure(s): this function computes the set of NFA states that are reachable from the NFA state s by going through ε-transitions only. Note that by definition s always belongs to ε-closure(s) since s is always reachable from s by going through zero ε-transitions.
- ε-closure(S): this function computes the set of NFA states that are reachable from any of the NFA states s in the set of NFA states S by going through ε-transitions only. This function is simply a generalization of the previous function to a set of states: $\varepsilon\text{-closure}(S) = \bigcup_{s \in S} \varepsilon\text{-closure}(s)$.
- $move_N(S, c)$: this function computes the set of NFA states to which the NFA can move when seeing input symbol c while in any of the NFA states s in the set of NFA states S. This function is simply a generalization of the usual $move_N$ transition function of an NFA to a set of states: $move_N(S, c) = \bigcup_{s \in S} move_N(s, c)$.

For example, given the following NFA:



we then have ε-closure$(0) = \{0, 1, 4, 5\}$. State 0 is in ε-closure(0) because it can be reached from state 0 through zero ε-transitions. States 1 and 4 are in ε-closure(0) because they can be reached from state 0 through one ε-transition, and state 5 is in ε-closure(0) because it can be reached from state 0 through two ε-transitions (by first going through state 1). Similarly, ε-closure$(1) = \{1, 5\}$, ε-closure$(2) = \{2, 3\}$, ε-closure$(3) = \{3\}$, ε-closure$(4) = \{4\}$, ε-closure$(5) = \{5\}$, and ε-closure$(6) = \{6\}$. Then, for example, ε-closure$(\{0, 2\}) = \varepsilon$-closure(0) $\cup$ ε-closure$(2) = \{0, 1, 4, 5\} \cup \{2, 3\} = \{0, 1, 2, 3, 4, 5\}$. Also, for example, $move_N(\{1, 4\}, \mathtt{a}) = move_N(1, \mathtt{a}) \cup move_N(4, \mathtt{a}) = \{2\} \cup \{1, 5\} = \{1, 2, 5\}$.

The reason we need to have the ε-closure function is because the DFAs we are going to construct will need to simulate the ε-transitions done by the corresponding NFAs (as indicated in the second example above). The reason we then need to generalize the ε-closure and $move_N$ functions to take a set of NFA states as argument (instead of just a single NFA state) is because each state of the DFAs we are going to construct will have to simulate a set of NFA states.

With these function definitions in mind, we can now look at how to compute a DFA equivalent to a given NFA. This method is called the *subset construction algorithm*. It takes as input an NFA $<S_N, \Sigma, move_N, n_0, F_N>$ and computes as output a DFA $<S_D, \Sigma, move_D, d_0, F_D>$. The algorithm has three steps and relies on DFA states being either unmarked (yet to be processed by the algorithm) or marked (already processed by the algorithm). Each DFA state d_i that the algorithm computes is a *set* of NFA states. The algorithm is then as follows:

1. Compute $d_0 = \varepsilon$-closure(n_0) and add the unmarked d_0 to S_D.

2. While there is an unmarked DFA state $d_i \in S_D$, do the following:
   (a) For each input symbol $\mathtt{a} \in \Sigma$, do the following:
      i. Compute the DFA state $d_j = \varepsilon\text{-closure}(\mathit{move}_N(d_i, \mathtt{a}))$.
      ii. If $d_j \neq \emptyset$ and $d_j \notin S_D$ then add the unmarked d_j to S_D.
      iii. If $d_j \neq \emptyset$ then define $\mathit{move}_D(d_i, \mathtt{a}) = d_j$.
   (b) Mark d_i in S_D.
3. For each $n_f \in F_N$, if $n_f \in d_k$ (for some k) then $d_k \in F_D$.

The first step starts the algorithm by computing the start state of the new DFA. The start state d_0 of the DFA is simply the ε-closure of the start state n_0 of the NFA (i.e. the start state of the DFA corresponds to the set of NFA states that are reachable from the start state of the NFA through ε-transitions only). This corresponds to the fact that the NFA can be in multiple possible states right from the start (as was in the case in the second example above).

Since the start state d_0 of the DFA is initially unmarked, it is then processed (and therefore marked, see step 2-b in the algorithm) in the second step. In fact every DFA state is unmarked when it is created and added to the set S_D of DFA states (see step 2-a-ii) so every DFA state will at some point be processed (once and only once) by the second step of the algorithm and then marked (step 2-b).

The way the second step in the algorithm processes an unmarked DFA state d_i is by considering transitions from that state d_i on every possible input symbol $\mathtt{a}$ (step 2-a). To do this (in step 2-a-i) the algorithm takes d_i (which is a set of NFA states), uses the generalized move_N transition function to compute the set of all NFA states that are reachable from the NFA states in d_i through a transition on $\mathtt{a}$ (the NFA might be in any of the states in d_i, so the DFA should simulate all possible transitions on $\mathtt{a}$ starting from any of those states by using the generalized move_N transition function), and then computes the ε-closure of that resulting set of NFA states to take into account the fact that the NFA can always make ε-transitions without consuming any input. The result of the ε-closure is then a set of NFA states which is named d_j. If that state does not already exist then it is added to the set S_D of DFA states (step 2-a-ii). Since it is added unmarked, we know it will be processed in step 2 of the algorithm some time later. From step 2-a-i we also know there should be in the DFA a transition on $\mathtt{a}$ from DFA state d_i to DFA state d_j (step 2-a-iii, which corresponds to adding an edge labeled with $\mathtt{a}$ going from node d_i to node d_j in the diagram representing the DFA).

As step 2 of the algorithm processes unmarked states and marks them, new unmarked DFA states might be created, which in turn are processed by step 2 of the algorithm, and so on like this until no more DFA states are created and all the existing DFA states have been processed.

Once the second step is over, the third and last step then determines which states of the DFA are final states. Remember that every DFA state d_k is a set of NFA states. A state d_k is then a final state of the DFA (i.e. is in the set

F_D of final states of the DFA) if it contains a final state n_f of the NFA (i.e. it contains a state n_f which is in the set F_N of final states of the NFA).

A few things should be noted regarding the subset construction algorithm:

- The algorithm works for all NFAs, not just for the kind of NFAs that are the result of using Thompson's construction method.

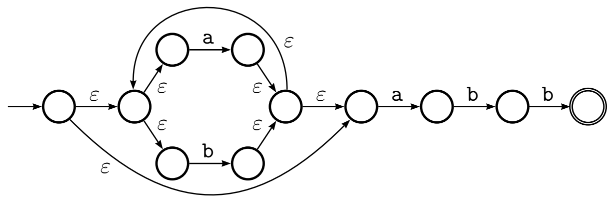
- Since every NFA can be converted into an equivalent DFA, this proves that NFAs are *not* more powerful than DFAs: the nondeterminism of NFAs does not give them any extra computing power compared to DFAs. Since every NFA has a equivalent DFA and since every DFA is also an NFA, this shows that NFAs and DFAs have the exact same computing power, which is not obvious.

- As we will see with a few examples below, the DFA computed by the subset construction algorithm is not always the smallest possible DFA that can be constructed for a given NFA. There are extra algorithms that can be applied to a DFA to minimize its size (the number of its states) but we will not look at such algorithms in this course. See the various compiler and automata theory textbooks for details about DFA minimization if you are interested. Minimizing the size of DFAs is a good idea though, because the smaller a DFA is the faster the lexer will be when using that DFA to try to match some input.

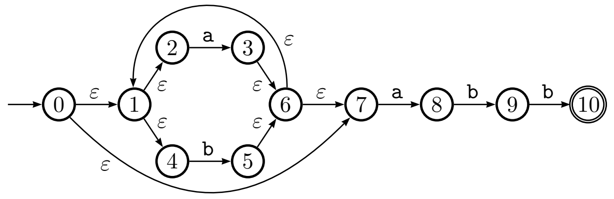
- If the NFA has n states then 2^n different sets of NFA states can be created (every state of the NFA can be either present or absent in a set of NFA states, hence the power of 2). Since every state of the DFA corresponds to a set of NFA states, this means that the DFA computed by the subset construction algorithm can potentially have 2^n different states! Remember that the number of states of an NFA constructed using Thompson's method is at most twice the number of symbols and operators in the original regular expression. This means that a small regular expression can lead to a small NFA which can lead to an enormous DFA! Obviously an exponential explosion in the number of DFA states is undesirable if we want to use that DFA in a lexer to match input: even if using such a DFA might still be in theory fast (the lexer still only needs to keep track of a single DFA state at any point in time) the amount of memory required to store the DFA might make using the DFA impossible. It is in fact possible to write regular expressions that lead to such exponential blowup when using the subset construction algorithm (see for example page 128 in the first edition of the Dragon book) but the kind of regular expressions that results in such exponential behavior of the algorithm is never seen in practice when using regular expressions to define the tokens of a programming language. So, while such exponential problem of the subset construction algorithm might happen in theory, in practice it it does not happen and the DFAs constructed by the subset algorithm are almost always more or less of the same size as the corresponding NFA.

- Processing a DFA state in step 2 of the subset construction algorithm might result in the creation of new DFA states. Processing these new states might in turn result in the creation of new DFA states, and so on. So it is not entirely clear whether the algorithm always terminates or not. It could potentially go on for ever, always creating new DFA states. Fortunately, as we have just seen, a DFA constructed using the subset construction algorithm can have only up to 2^n states when the original NFA has n states. Since every DFA state is only processed once in step 2 of the algorithm (a state goes from unmarked to marked, but never from marked to unmarked) this means that step 2 of the algorithm can happen at most 2^n times. The algorithm is therefore guaranteed to always terminate.

We can now look at a few examples to see how the subset construction algorithm actually works in practice. We start with the NFA we created using Thompson's construction method for the regular expression (`a`|`b`)*`abb`:



The first thing we do is to number the states:



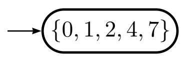
Applying the subset construction algorithm to this NFA is going to require us to compute the ε-closure of many states, so to simplify the process we compute the ε-closure of every state in advance. The ε-closure of an NFA state s is the set of NFA states that are reachable from s by going through any number of ε-transitions (and ε-transitions only). So, for example, ε-closure(0) contains the state 0 itself, since the state 0 of the NFA can obviously be reached from state 0 by going through zero ε-transitions (ε-closure(s) always contains s itself, as we have indicated before). The set ε-closure(0) also contains state 1, since state 1 is reachable from state 0 through one ε-transition. Similarly state 7 is reachable from state 0 through one ε-transition, so state 7 is in ε-closure(0) too. In turn states 2 and 4 are in ε-closure(0) since both these states can be reached from state 0 by going through ε-transitions only, first from state 0 to state 1 and then from state 1 to either state 2 or state 4. So ε-closure(0) $= \{0, 1, 2, 4, 7\}$.

Similarly, ε-closure(3) $= \{1, 2, 3, 4, 6, 7\}$. State 3 is obviously in ε-closure(3), state 6 can be reached from state 3 through a single ε-transition, states 1 and 7 can be reached from state 3 through two ε-transitions (by going through state 6 first), and states 2 and 4 can be reached from state 3 through three ε-transitions (by going first through state 6 then state 1).

By computing the ε-closure of every state in the NFA in this manner, we get the following table:

NFA State s	ε-closure(s)
0	{0,1,2,4,7}
1	{1,2,4}
2	{2}
3	{1,2,3,4,6,7}
4	{4}
5	{1,2,4,5,6,7}
6	{1,2,4,6,7}
7	{7}
8	{8}
9	{9}
10	{10}

We can now start the subset construction algorithm itself. The first step is to compute the start state d_0 of the DFA by computing the ε-closure of the start state of the NFA. We have computed just above that $\varepsilon\text{-closure}(0) = \{0, 1, 2, 4, 7\}$ so we create the start state of the DFA to be this set of NFA states:



This set {0,1,2,4,7} of NFA states in the start state of the DFA simulates the fact that the NFA can be in any of the states 0, 1, 2, 4, or 7 before reading any input.

Since the DFA's start state is unmarked, step 2 of the algorithm will now process this state. To do this, we have to consider transitions from this state on every possible alphabet symbol, which is this example are `a` and `b`. We start we `a`. To compute the DFA transition from the start state of the DFA on symbol `a`, we have to find all the possible NFA transitions on symbol `a` from any of the NFA states in the set of NFA states represented by the start state of the DFA. Since the start state of the DFA represents the set of NFA states $\{0, 1, 2, 4, 7\}$, we have to consider all the possible NFA transitions on symbol `a` from any of the NFA states 0, 1, 2, 4, or 7. To do this, we just use the transition function $move_N$ of the NFA on each NFA state 0, 1, 2, 4, or 7:

NFA State s	$move_N(s, \texttt{a})$
$0 \overset{\texttt{a}}{\to}$	$\emptyset$
$1 \overset{\texttt{a}}{\to}$	$\emptyset$
$2 \overset{\texttt{a}}{\to}$	{3}
$4 \overset{\texttt{a}}{\to}$	$\emptyset$
$7 \overset{\texttt{a}}{\to}$	{8}

Only the states 2 and 7 of the NFA have transitions on symbol a: state 2 to state 3, and state 7 to state 8. From this we conclude that $move_N(\{0,1,2,4,7\}, \mathtt{a}) = \{3,8\}$, by simply computing the union of the different sets $move_N(s, \mathtt{a})$ we just computed:

NFA State s	$move_N(s, \mathtt{a})$
$0 \overset{\mathtt{a}}{\to}$	$\emptyset$
$1 \overset{\mathtt{a}}{\to}$	$\emptyset$
$2 \overset{\mathtt{a}}{\to}$	$\{3\}$
$4 \overset{\mathtt{a}}{\to}$	$\emptyset$
$7 \overset{\mathtt{a}}{\to}$	$\{8\}$
$move_N(\{0,1,2,4,7\}, \mathtt{a})$	$\{3,8\}$

After the NFA has moved from state 2 to state 3, or from state 7 to state 8, the NFA can then make ε-transitions without consuming any input, so we have to take this into account by computing the ε-closure of the set $\{3,8\}$. From the table we computed above, we have $\varepsilon\text{-closure}(3) = \{1,2,3,4,6,7\}$ and $\varepsilon\text{-closure}(8) = \{8\}$. From this we conclude that $\varepsilon\text{-closure}(\{3,8\}) = \{1,2,3,4,6,7,8\}$, by simply computing the union of the two closure sets $\{1,2,3,4,6,7\}$ and $\{8\}$ we just computed:

NFA State s	$move_N(s, \mathtt{a})$
$0 \overset{\mathtt{a}}{\to}$	$\emptyset$
$1 \overset{\mathtt{a}}{\to}$	$\emptyset$
$2 \overset{\mathtt{a}}{\to}$	$\{3\}$
$4 \overset{\mathtt{a}}{\to}$	$\emptyset$
$7 \overset{\mathtt{a}}{\to}$	$\{8\}$
$move_N(\{0,1,2,4,7\}, \mathtt{a})$	$\{3,8\}$
$\varepsilon\text{-closure}(move_N(\{0,1,2,4,7\}, \mathtt{a}))$	$\{1,2,3,4,6,7,8\}$

We then define d_1 to be this new set $\{1,2,3,4,6,7,8\}$. In other words, $d_1 = \varepsilon\text{-closure}(move_N(d_0, \mathtt{a}))$ (step 2-a-i of the subset construction algorithm). Since the set d_1 is not empty and we do not yet have such a state in the DFA, we add d_1 to the set S_D of the DFA (step 2-a-ii of the algorithm):

$$\begin{tikzcd} {} \arrow[r] & \boxed{\{0,1,2,4,7\}} & \boxed{\{1,2,3,4,6,7,8\}} \end{tikzcd}$$

In step 2-a-iii we then define $move_D(d_0, \mathtt{a}) = d_1$. This means that the DFA moves from state d_0 to state d_1 on input a. This corresponds to the NFA moving on input a from one of the possible NFA states 0, 1, 2, 4, or 7, to one of the possible NFA states 1, 2, 3, 4, 6, 7, or 8. From the point of view of the diagram representing the DFA we are constructing, this corresponds to adding a transition from d_0 to d_1 on symbol a:

$$\begin{tikzcd} {} \arrow[r] & \boxed{\{0,1,2,4,7\}} \arrow[r, "\mathtt{a}"] & \boxed{\{1,2,3,4,6,7,8\}} \end{tikzcd}$$

We have just considered the transition from the start state of the DFA on input symbol `a`. We now have to do the same thing for input symbol `b`:

NFA State s	$move_N(s, \texttt{b})$
$0 \overset{\texttt{b}}{\rightarrow}$	$\emptyset$
$1 \overset{\texttt{b}}{\rightarrow}$	$\emptyset$
$2 \overset{\texttt{b}}{\rightarrow}$	$\emptyset$
$4 \overset{\texttt{b}}{\rightarrow}$	$\{5\}$
$7 \overset{\texttt{b}}{\rightarrow}$	$\emptyset$

$move_N(\{0,1,2,4,7\}, \texttt{b})$	$\{5\}$
ε-closure($move_N(\{0,1,2,4,7\}, \texttt{b})$)	$\{1,2,4,5,6,7\}$

Since the set $\{1,2,4,5,6,7\}$ does not yet exist in our DFA, we add a new state d_2 to our DFA (step 2-a-ii of the algorithm). We also add a transition from d_0 to d_2 on input symbol `b` (step 2-a-iii of the algorithm):

$$\begin{tikzcd}
& & \{1,2,4,5,6,7\} \\
{} \arrow[r] & \{0,1,2,4,7\} \arrow[r, "\texttt{a}"] \arrow[ur, "\texttt{b}"] & \{1,2,3,4,6,7,8\}
\end{tikzcd}$$

At this point we have consider the start state d_0 of the DFA for transitions on every possible input symbol. We are therefore done processing this state (the “for” loop of step 2-a of the algorithm stops) and we mark the state d_0 as processed (step 2-b of the algorithm):

$$\begin{tikzcd}
& & \{1,2,4,5,6,7\} \\
{} \arrow[r] & \{0,1,2,4,7\}^{*} \arrow[r, "\texttt{a}"] \arrow[ur, "\texttt{b}"] & \{1,2,3,4,6,7,8\}
\end{tikzcd}$$

We now repeat the whole process for state d_1, which is currently unmarked. We first consider the possible transitions on input symbol `a`. The process is always the same: compute the possible transitions, take the union of the results, and compute the ε-closure of the union:

NFA State s	$move_N(s, \texttt{a})$
$1 \overset{\texttt{a}}{\rightarrow}$	$\emptyset$
$2 \overset{\texttt{a}}{\rightarrow}$	$\{3\}$
$3 \overset{\texttt{a}}{\rightarrow}$	$\emptyset$
$4 \overset{\texttt{a}}{\rightarrow}$	$\emptyset$
$6 \overset{\texttt{a}}{\rightarrow}$	$\emptyset$
$7 \overset{\texttt{a}}{\rightarrow}$	$\{8\}$
$8 \overset{\texttt{a}}{\rightarrow}$	$\emptyset$

$move_N(\{1,2,3,4,6,7,8\}, \texttt{a})$	$\{3,8\}$
ε-closure($move_N(\{1,2,3,4,6,7,8\}, \texttt{a})$)	$\{1,2,3,4,6,7,8\}$

Since we already have in the DFA a state for the set $\{1, 2, 3, 4, 6, 7, 8\}$ of NFA states, we do not need to add a new state to the DFA. In fact the computation we just did shows that $move_D(d_1, \texttt{a}) = d_1$, so there is now a loop in the DFA:

$$\begin{tikzcd}
 & & \{1,2,4,5,6,7\} \\
{} \arrow[r] & \{0,1,2,4,7\}^{*} \arrow[ur, "\texttt{b}"] \arrow[r, "\texttt{a}"] & \{1,2,3,4,6,7,8\} \arrow[loop below, "\texttt{a}"]
\end{tikzcd}$$

We now consider transitions from d_1 on input symbol $\texttt{b}$:

NFA State s	$move_N(s, \texttt{b})$
$1 \xrightarrow{\texttt{b}}$	$\emptyset$
$2 \xrightarrow{\texttt{b}}$	$\emptyset$
$3 \xrightarrow{\texttt{b}}$	$\emptyset$
$4 \xrightarrow{\texttt{b}}$	$\{5\}$
$6 \xrightarrow{\texttt{b}}$	$\emptyset$
$7 \xrightarrow{\texttt{b}}$	$\emptyset$
$8 \xrightarrow{\texttt{b}}$	$\{9\}$
$move_N(\{1, 2, 3, 4, 6, 7, 8\}, \texttt{b})$	$\{5, 9\}$
ε-closure$(move_N(\{1, 2, 3, 4, 6, 7, 8\}, \texttt{b}))$	$\{1, 2, 4, 5, 6, 7, 9\}$

So we add a new state d_3 and a new transition to the DFA:

$$\begin{tikzcd}
 & & \{1,2,4,5,6,7\} & \\
{} \arrow[r] & \{0,1,2,4,7\}^{*} \arrow[ur, "\texttt{b}"] \arrow[r, "\texttt{a}"] & \{1,2,3,4,6,7,8\} \arrow[loop below, "\texttt{a}"] \arrow[r, "\texttt{b}"] & \{1,2,4,5,6,7,9\}
\end{tikzcd}$$

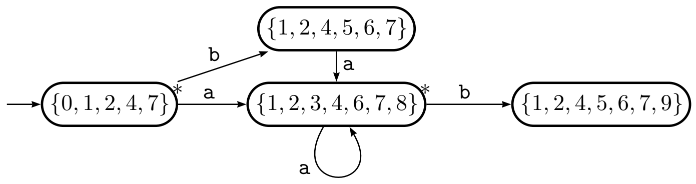
and state d_1 is now marked as processed:

$$\begin{tikzcd}
 & & \{1,2,4,5,6,7\} & \\
{} \arrow[r] & \{0,1,2,4,7\}^{*} \arrow[ur, "\texttt{b}"] \arrow[r, "\texttt{a}"] & \{1,2,3,4,6,7,8\}^{*} \arrow[loop below, "\texttt{a}"] \arrow[r, "\texttt{b}"] & \{1,2,4,5,6,7,9\}
\end{tikzcd}$$

We now process state d_2 on input $\texttt{a}$:

NFA State s	$move_N(s, \texttt{a})$
$1 \overset{\texttt{a}}{\rightarrow}$	$\emptyset$
$2 \overset{\texttt{a}}{\rightarrow}$	$\{3\}$
$4 \overset{\texttt{a}}{\rightarrow}$	$\emptyset$
$5 \overset{\texttt{a}}{\rightarrow}$	$\emptyset$
$6 \overset{\texttt{a}}{\rightarrow}$	$\emptyset$
$7 \overset{\texttt{a}}{\rightarrow}$	$\{8\}$
$move_N(\{1,2,4,5,6,7\}, \texttt{a})$	$\{3,8\}$
ε-closure($move_N(\{1,2,4,5,6,7\}, \texttt{a})$)	$\{1,2,3,4,6,7,8\}$

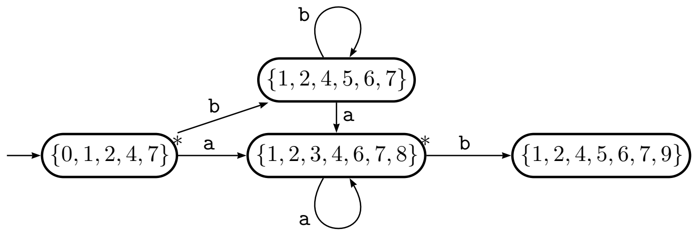
so we just add a transition on `a` from state d_2 to state d_1:



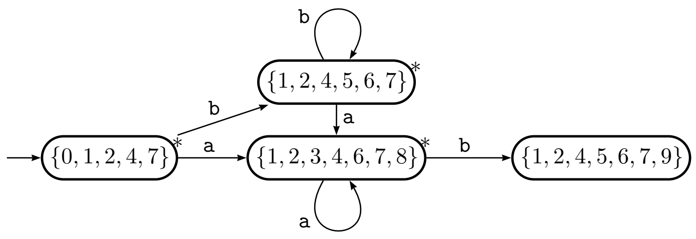
Processing d_2 on input `b` gives:

NFA State s	$move_N(s, \texttt{b})$
$1 \overset{\texttt{b}}{\rightarrow}$	$\emptyset$
$2 \overset{\texttt{b}}{\rightarrow}$	$\emptyset$
$4 \overset{\texttt{b}}{\rightarrow}$	$\{5\}$
$5 \overset{\texttt{b}}{\rightarrow}$	$\emptyset$
$6 \overset{\texttt{b}}{\rightarrow}$	$\emptyset$
$7 \overset{\texttt{b}}{\rightarrow}$	$\emptyset$
$move_N(\{1,2,4,5,6,7\}, \texttt{b})$	$\{5\}$
ε-closure($move_N(\{1,2,4,5,6,7\}, \texttt{b})$)	$\{1,2,4,5,6,7\}$

so there is a loop from state d_2 to itself on input `b`:



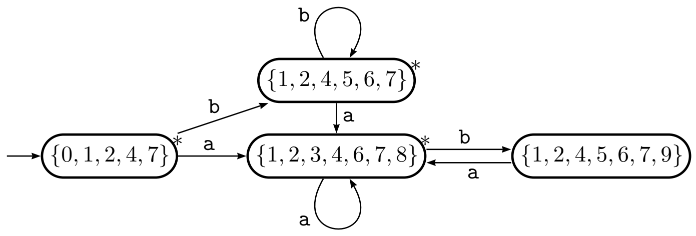
and state d_2 has now been completely processed:



We now process state d_3 on input `a`:

NFA State s	$move_N(s, \texttt{a})$
$1 \xrightarrow{\texttt{a}}$	$\emptyset$
$2 \xrightarrow{\texttt{a}}$	$\{3\}$
$4 \xrightarrow{\texttt{a}}$	$\emptyset$
$5 \xrightarrow{\texttt{a}}$	$\emptyset$
$6 \xrightarrow{\texttt{a}}$	$\emptyset$
$7 \xrightarrow{\texttt{a}}$	$\{8\}$
$9 \xrightarrow{\texttt{a}}$	$\emptyset$
$move_N(\{1, 2, 4, 5, 6, 7, 9\}, \texttt{a})$	$\{3, 8\}$
ε-closure($move_N(\{1, 2, 4, 5, 6, 7, 9\}, \texttt{a})$)	$\{1, 2, 3, 4, 6, 7, 8\}$

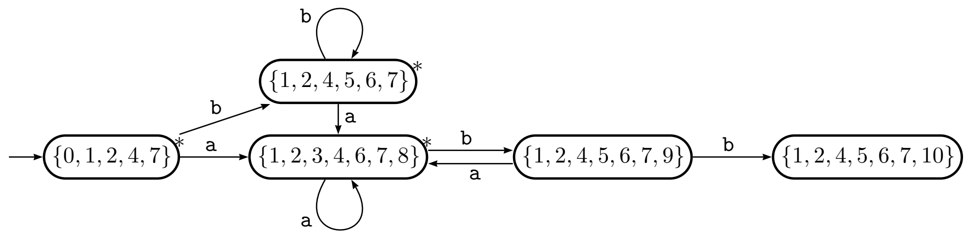
so we add a transition on `a` from state d_3 to state d_1:



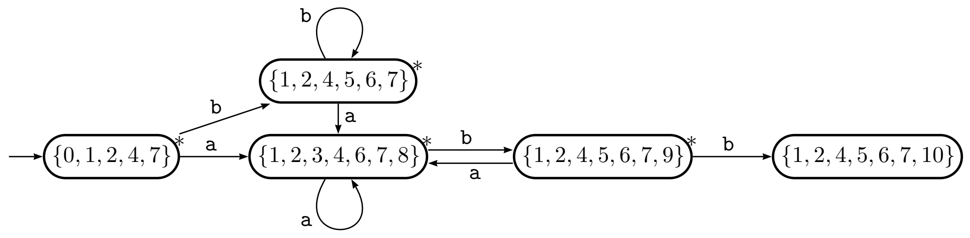
Processing state d_3 on input `b`:

NFA State s	$move_N(s, \texttt{b})$
$1 \xrightarrow{\texttt{b}}$	$\emptyset$
$2 \xrightarrow{\texttt{b}}$	$\emptyset$
$4 \xrightarrow{\texttt{b}}$	$\{5\}$
$5 \xrightarrow{\texttt{b}}$	$\emptyset$
$6 \xrightarrow{\texttt{b}}$	$\emptyset$
$7 \xrightarrow{\texttt{b}}$	$\emptyset$
$9 \xrightarrow{\texttt{b}}$	$\{10\}$
$move_N(\{1, 2, 4, 5, 6, 7, 9\}, \texttt{b})$	$\{5, 10\}$
ε-closure($move_N(\{1, 2, 4, 5, 6, 7, 9\}, \texttt{b})$)	$\{1, 2, 4, 5, 6, 7, 10\}$

so we add a new state d_4 to the DFA and a transition on `b` from d_3 to d_4:



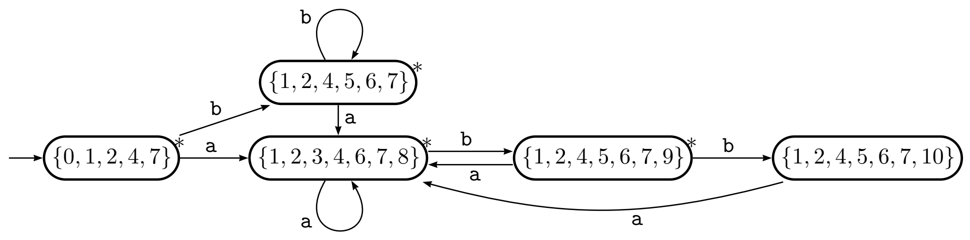
and we now mark d_3 as processed:



We now process d_4 on input `a`:

NFA State s	$move_N(s, \texttt{a})$
$1 \xrightarrow{\texttt{a}}$	$\emptyset$
$2 \xrightarrow{\texttt{a}}$	$\{3\}$
$4 \xrightarrow{\texttt{a}}$	$\emptyset$
$5 \xrightarrow{\texttt{a}}$	$\emptyset$
$6 \xrightarrow{\texttt{a}}$	$\emptyset$
$7 \xrightarrow{\texttt{a}}$	$\{8\}$
$10 \xrightarrow{\texttt{a}}$	$\emptyset$
$move_N(\{1, 2, 4, 5, 6, 7, 10\}, \texttt{a})$	$\{3, 8\}$
ε-closure($move_N(\{1, 2, 4, 5, 6, 7, 10\}, \texttt{a})$)	$\{1, 2, 3, 4, 6, 7, 8\}$

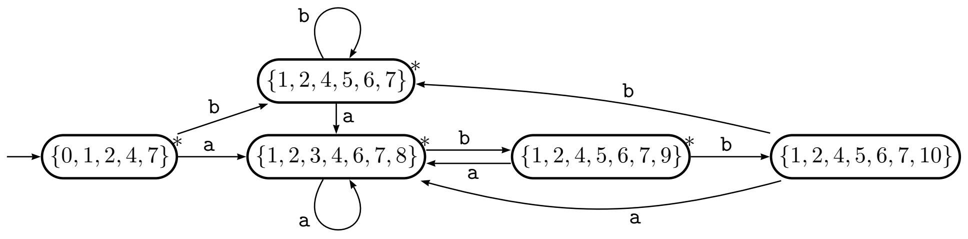
so we add a transition from d_4 to d_1 on input `a`:



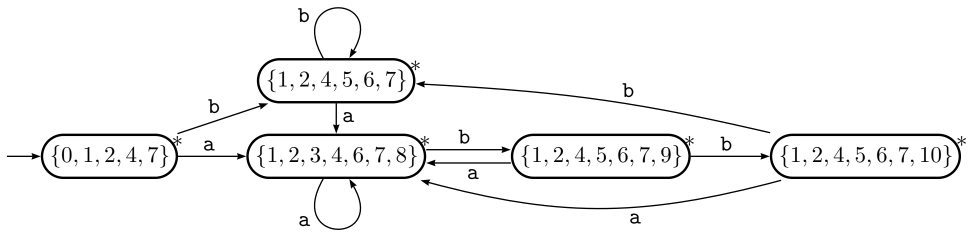
We then process state d_4 on input `b`:

NFA State s	$move_N(s, \texttt{b})$
$1 \xrightarrow{\texttt{b}}$	$\emptyset$
$2 \xrightarrow{\texttt{b}}$	$\emptyset$
$4 \xrightarrow{\texttt{b}}$	$\{5\}$
$5 \xrightarrow{\texttt{b}}$	$\emptyset$
$6 \xrightarrow{\texttt{b}}$	$\emptyset$
$7 \xrightarrow{\texttt{b}}$	$\emptyset$
$10 \xrightarrow{\texttt{b}}$	$\emptyset$
$move_N(\{1,2,4,5,6,7,9\}, \texttt{b})$	$\{5\}$
ε-closure$(move_N(\{1,2,4,5,6,7,9\}, \texttt{b}))$	$\{1,2,4,5,6,7\}$

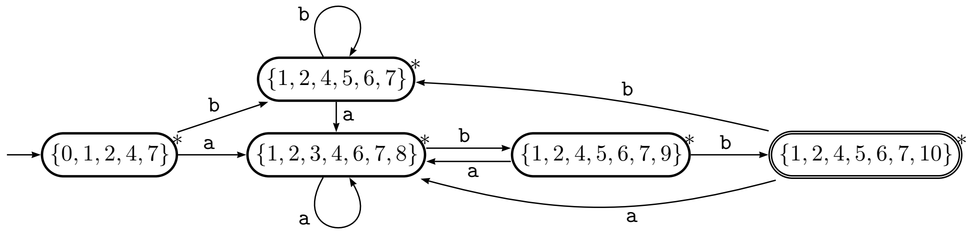
so we add a transition from d_4 to d_2 on input `b`:



and we then mark d_4 as processed:

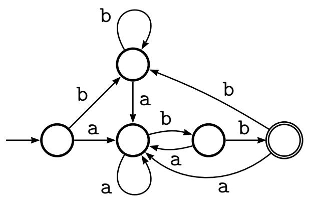


At this point all states of the DFA have been processes, so the "while" loop in step 2 of the algorithm is over. We now have to do the third and last step of the algorithm: compute the final states of the DFA. A state of the DFA is a final state if and only if it contains a final state of the NFA. The NFA we use in this example has only one final state, which is state 10. The only DFA state that contains NFA state 10 is d_4, so we mark this state d_4 as a final state:



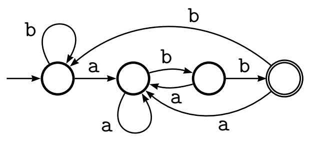
and we are now done.

Here is the same DFA without the sets of NFA states being explicit:



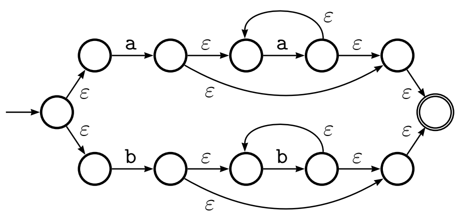
There are a few things to note about this DFA:

- If you test this DFA on various input strings, you will see that it accepts or rejects exactly the same strings as the original NFA. This is what we expect, since the DFA is supposed to simulate everything the NFA does. If there were some input string for which the behaviors of the NFA and of the DFA were different then this would be a clear sign that something is wrong in the way we constructed the DFA.
- The original NFA has 11 states and the DFA we just constructed has 5 states. This shows that the exponential blowup in the number of states that is one of the theoretical problems of the subset construction algorithm did not happen in this case. In fact it is extremely rare for such exponential blowup to happen. In practice the subset construction algorithm constructs DFAs which are more or less of the same size as the NFA, as is the case with our example here.
- The DFA created by the subset construction algorithm is not always the smallest possible DFA that can be constructed for a given NFA. As we have seen in the section about nondeterministic finite automata above, another DFA that recognizes the language described by the regular expression $(\texttt{a}|\texttt{b})^*\texttt{abb}$ is:

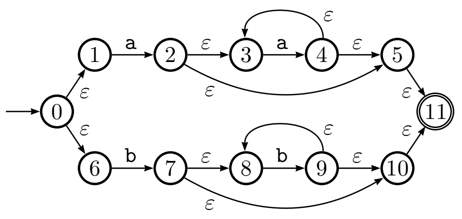
  

  and this DFA has one fewer states than the one we just constructed (the subset construction algorithm would have created the same DFA if it had somehow realized it can merge states d_0 and d_2 into a single start state).

Here is another example of how to use the subset construction algorithm. This time we want to construct the DFA corresponding to the following NFA, which we built earlier from the regular expression $\texttt{aa}^*|\texttt{bb}^*$ using Thompson's construction method:



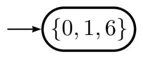
First we number the states of the NFA:



and compute the table for the ε-closure function:

NFA State s	ε-closure(s)
0	{0,1,6}
1	{1}
2	{2,3,5,11}
3	{3}
4	{3,4,5,11}
5	{5,11}
6	{6}
7	{7,8,10,11}
8	{8}
9	{8,9,10,11}
10	{10,11}
11	{11}

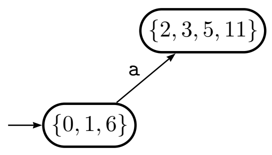
We can now start creating the corresponding DFA using the subset construction algorithm. First we create the start state of the DFA by using ε-closure(0), which we just computed above:



We then consider the possible transitions from this state on input symbol `a`:

NFA State s	$move_N(s, \mathtt{a})$
$0 \overset{\mathtt{a}}{\to}$	$\emptyset$
$1 \overset{\mathtt{a}}{\to}$	$\{2\}$
$6 \overset{\mathtt{a}}{\to}$	$\emptyset$
$move_N(\{0,1,6\}, \mathtt{a})$	$\{2\}$
ε-closure($move_N(\{0,1,6\}, \mathtt{a})$)	$\{2,3,5,11\}$

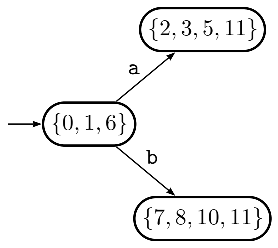
so we add a new state to the DFA and a new transition on `a`:



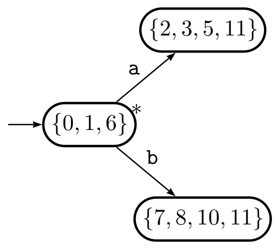
We then consider the possible transitions from the start state of the DFA in input `b`:

NFA State s	$move_N(s, \texttt{b})$
$0 \xrightarrow{\texttt{b}}$	$\emptyset$
$1 \xrightarrow{\texttt{b}}$	$\emptyset$
$6 \xrightarrow{\texttt{b}}$	$\{7\}$
$move_N(\{0, 1, 6\}, \texttt{b})$	$\{7\}$
ε-closure($move_N(\{0, 1, 6\}, \texttt{b})$)	$\{7, 8, 10, 11\}$

so we add a new state to the DFA and a new transition on `b`:



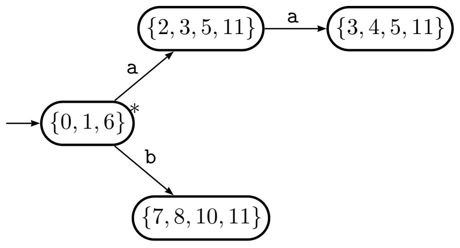
and the start state has now been completely processed:



We now process the second DFA state on input `a`:

NFA State s	$move_N(s, \texttt{a})$
$2 \xrightarrow{\texttt{a}}$	$\emptyset$
$3 \xrightarrow{\texttt{a}}$	$\{4\}$
$5 \xrightarrow{\texttt{a}}$	$\emptyset$
$11 \xrightarrow{\texttt{a}}$	$\emptyset$
$move_N(\{2, 3, 5, 11\}, \texttt{a})$	$\{4\}$
ε-closure($move_N(\{2, 3, 5, 11\}, \texttt{a})$)	$\{3, 4, 5, 11\}$

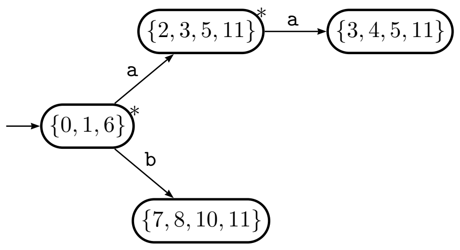
and again add a new state and a new transition:



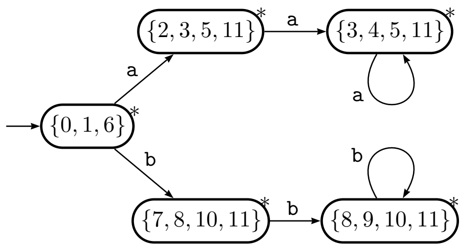
When we process the second state on input symbol `b` we get:

NFA State s	$move_N(s, \texttt{b})$
$2 \xrightarrow{\texttt{b}}$	$\emptyset$
$3 \xrightarrow{\texttt{b}}$	$\emptyset$
$5 \xrightarrow{\texttt{b}}$	$\emptyset$
$11 \xrightarrow{\texttt{b}}$	$\emptyset$
$move_N(\{2, 3, 5, 11\}, \texttt{b})$	$\emptyset$
ε-closure($move_N(\{2, 3, 5, 11\}, \texttt{b})$)	$\emptyset$

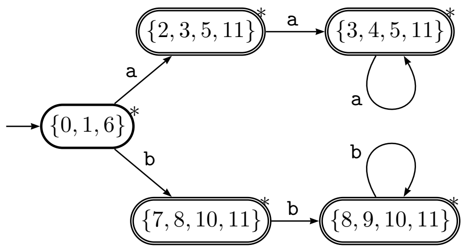
Since the set we just computed is empty there is nothing to do: no new state and no new transition is added to the DFA. We are then done processing the second state:



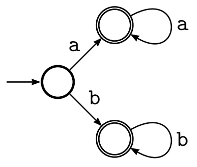
Similarly we process the other states of the DFA until we get:



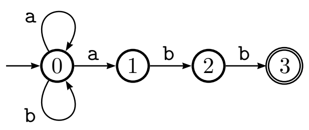
At this point all the states in the DFA have been processes and step 2 of the subset construction algorithm is over. We now have to do the third and last step of the algorithm: compute the final states of the DFA. A state of the DFA is a final state if and only if it contains a final state of the NFA. The NFA we use in this example has only one final state, which is state 11. All four states of the DFA that contain NFA state 11 are therefore marked as final states of the DFA:



and we are now done. Note again that this DFA is not the smallest DFA that can be constructed for the regular expression `aa`*|`bb`*. Here is a smaller DFA for the same regular expression:



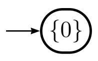
To finish with the subset construction algorithm, here is an example of applying the algorithm to an NFA which is not the result of using Thompson's construction method. We have seen before that a simple NFA recognizing the strings described by the regular expression (`a`|`b`)*`abb` is:



Since this NFA does not have ε-transitions, the table for the ε-closure function is very simple:

NFA State s	ε-closure(s)
0	{0}
1	{1}
2	{2}
3	{3}

We can then use ε-closure(0) to construct the start state of an equivalent DFA:

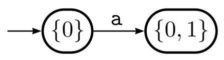


We can then compute the transition from this start state on symbol `a`:

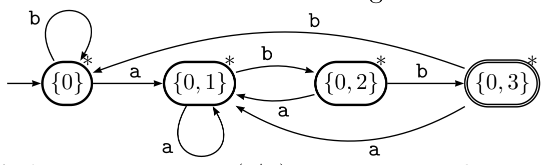
NFA State s	$move_N(s, \mathtt{a})$
$0 \overset{\mathtt{a}}{\rightarrow}$	{0,1}
$move_N(\{0\}, \mathtt{a})$	$\{0, 1\}$
ε-closure($move_N(\{0\}, \mathtt{a})$)	$\{0, 1\}$

Note that in this table the entry for $move_N(0, \texttt{a})$ contains two NFA states: 0 and 1. This is because there are two transitions on `a` going out of state 0 in the NFA. Note that this is the first such table in these lecture notes where an entry has more than one NFA state in it. This is because Thompson's construction method guarantees that the NFAs created using that method only have at most one transition for a given input symbol out of any given state. The only case in which a state can have more than one transition out of it for the same symbol is when the symbol is ε, i.e. for ε-transitions. This in turns implies that the sets $move_N(s, c)$ we have computed so far for NFAs created using Thompson's method only have had at most one NFA state in them (i.e. they are either the empty set or a singleton set). If you apply the subset construction algorithm to an NFA that was created using Thompson's construction method and you end up with a set $move_N(s, c)$ that has more than one NFA state in it, then you know you made a mistake. Here the NFA we are applying the subset construction algorithm to was not created Thompson's construction method, so some sets $move_N(s, c)$ might then contain more than one NFA state.

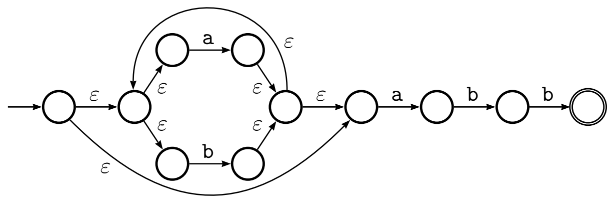
We then get the following DFA:



Processing then continues as usual until we get the DFA:



This is the DFA for the language $(\texttt{a}|\texttt{b})^*\texttt{abb}$ that we first saw in the section on nondeterministic finite automata at the start of these lecture notes. It is different from the DFA we created above when applying the subset construction algorithm to the NFA that was the result of using Thompson's construction method:

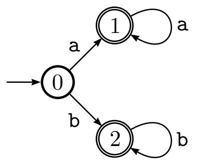


This shows that, starting from two different NFAs for the same regular expression, the subset construction algorithm might produce two different DFAs.

From Deterministic Finite Automaton to C Code

Now that we know how to transform a regular expression into an NFA as well as how to transform an NFA into a DFA, the last step we should now take is to see how to transform a DFA into C code. That C code can then be used to check whether some input is matched by the original regular expression or not, which is what we are interested in: since tokens are described using regular expressions, a lexer can then use the C code derived from the regular expressions to determine whether some input string belongs to some token or not.

To understand how to convert a DFA into C code, let's look at a simple example. The following DFA recognizes input strings that are described by the regular expression `aa`*|`bb`*:



To write C code implementing this DFA, we simply define a variable `state` that keeps track of the current state of the DFA. That variable is initialized to store the number of the start state of the DFA, which is 0. The C code then has a `switch` statement, with one case for each state in the DFA. The code for each case simply looks at the next input symbol and then updates the `state` variable with a new state number, which is determined by looking at what the DFA is supposed to do when it sees such input symbol in such state. The `switch` statement is itself inside a `while` loop, to repeat the same process until all input has been consumed.

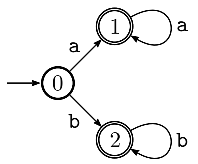
So for example the DFA above is implemented by the following C code:

```
int state = 0; /* DFA in start state */
char c;
while(1){
  switch(state){
    case 0: /* DFA in state 0 */
      c = nextchar();
      if(c == 'a'){
        state = 1; /* move from state 0 to state 1 on input 'a' */
      }else{
        if(c == 'b'){
          state = 2; /* move from state 0 to state 1 on input 'b' */
        }else{
          error_recovery(); /* DFA is stuck in state 0 */
        }
      }
      break;
    case 1: /* DFA in state 1 */
      c = nextchar();
      if(c == 'a'){
        state = 1; /* move from state 1 to state 1 on input 'a' */
      }else{
        retract_last_input(c); /* read one char too many */
        return SOMETOKEN;   /* returns token to parser */
      }
      break;
    case 2: /* DFA in state 2 */
      c = nextchar();
      if(c == 'b'){
        state = 2; /* move from state 2 to state 2 on input 'b' */
      }else{
        retract_last_input(c); /* read one char too many */
        return SOMETOKEN;
      }
      break;
  }
}
```

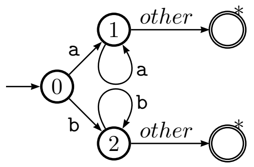
The C code starts with `state` set to `0`, which simulates the DFA being initially in state 0. The `case 0` is therefore executed when the code enters the `while` loop and the `switch` statement. The program then reads one input character using the `nextchar` function. If that character is `a` then `state` is set to `1`, to simulate the DFA going from state 0 to state 1 on reading the input symbol `a`. This then means that at the next iteration of the `while` loop the code for `case 1` will be executed. If the character just read is `b` then `state` is set to `2`, to simulate the DFA going from state 0 to state 2 on reading the input symbol `b`. If the character just read is neither `a` nor `b` then a lexical error has been detected and some error recovery has to be done (see below).

When the code is in `case 1` (i.e. the DFA is in state 1), the code similarly reads the next character in the input and checks to see if it is the character `a` or not. If it is, then `state` gets the value `1` (which is a bit useless, since `state` already has the value `1`, but the assignment explicitly simulates the move of the DFA from state 1 to state 1 on input `a`). If the next character in the input is not `a`, then the code does not this time call the error recovery procedure. That is because state 1 of the DFA is a final state. This means that, when the DFA is in state 1 and suddenly sees some input which is not the symbol `a`, the DFA then knows it has reached the end of the lexeme it is currently matching. This corresponds to the "other" case we have discussed at the very beginning of these lecture notes. In that case the C code uses the procedure `retract_last_input` to put the last character read back into the input (to be processed later again when the parser calls the lexer again to identify the next lexeme in the input) and then returns (to the parser, presumably) a C constant `SOMETOKEN` to indicate that the C code has successfully identified in the input a lexeme that is matched by the regular expression $\mathtt{aa}^*|\mathtt{bb}^*$. Similarly for the `case 2` in the code.

If you look closely at the C code, at the way the code checks the variable `c` in each `case`, and at the way we use `retract_last_input`, you will realize that the C code above does not implement this DFA:



but rather implements the following one, which is the kind of DFA lexers actually use in practice (see our discussion of this topic at the beginning of these lecture notes, when we looked at the way lexers use DFAs to match input):



where the * corresponds to putting the last symbol read back into the input.

Transforming a DFA into C code is therefore quite simple:

- Use a `state` variable to keep track of the current state of the DFA.
- Have a `switch` statement with one `case` for each state in the DFA.
- In each `case`, read one input character, test the character, and modify the `state` variable to simulate the DFA moving through a transition to another state when reading that input symbol. Each transition in the DFA then corresponds to an assignment to the `state` variable in the C code.
- If there is no possible transition in the current state for the input character just read, and the current state is a final state, put the last character read back into the input and return a token: a lexeme has now been identified.
- If there is no possible transition in the current state for the input character just read, and the current state is not a final state, then a lexical error has been detected, so call the error recovery procedure.
- Wrap the whole `switch` statement inside a `while` loop so that the C code keeps reading input characters until it either matches a lexeme or detects a lexical error.

There is one more thing the code above should do, which is not shown: as the DFA moves from state to state (i.e. every time the `state` variable is modified in the C code above) the C code should store in an array of characters each character that is read (the character in the variable `c`). That way, when the C code returns a token to the parser, the array will always contain the complete lexeme corresponding to the token. This is what the `yytext` variable is used for in lexers generated using Lex or Flex: `yytext` is defined as an array of characters in which the lexer saves the input characters that it matches.

The C code above works well, but it is not very efficient:

- It reads one character at a time from the input, which is quite slow. What one really wants is to have C code that reads the input in big chunks (say, one kilobyte at a time) and then buffers that input in memory. Then reading a character from the input can be done by directly reading it from the buffer in memory, which is very fast.

- The code has to go around the `while` loop for each character read and then go through the `switch` statement to go to the right `case` in the code that simulates the current state of the DFA. Rather than doing this, it is more efficient to replace each `case` with a C label and then simply use `goto` statements to directly jump between the different cases. Then there is no need for the `state` variable anymore:

  ```
  char c;
  state0: /* DFA in state 0 */
    c = nextchar();
    if(c == 'a'){
      goto state1; /* move from state 0 to state 1 on input 'a' */
    }else{
      if(c == 'b'){
        goto state2; /* move from state 0 to state 1 on input 'b' */
      }else{
        error_recovery(); /* DFA is stuck in state 0 */
      }
    }
  state1: /* DFA in state 1 */
    c = nextchar();
    if(c == 'a'){
      goto state1; /* move from state 1 to state 1 on input 'a' */
    }else{
      retract_last_input(c); /* read one char too many */
      return SOMETOKEN;   /* returns token to parser */
    }
  state2: /* DFA in state 2 */
    c = nextchar();
    if(c == 'b'){
      goto state2; /* move from state 2 to state 2 on input 'b' */
    }else{
      retract_last_input(c); /* read one char too many */
      return SOMETOKEN;
    }
  }
  ```

  In this C code there is one label for each state in the DFA, and one `goto` for each transition in the DFA. This is one of the few cases in programming where using `goto` statements is a good idea, because then the shape of the code will exactly match the shape of the DFA.

We will not look in more details at these issues in this course, but you should be aware of them if one day you decide to implement a DFA by hand. Here we are content with just explaining the general idea of how to implement a DFA in C code.

The code above that implements a DFA is quite simple, but here is another way to implement the same DFA in C which is even simpler. In this version of the code we create a C array that stores the transition table for the *move* transition function of the DFA:

	`a`	`b`	other
0	1	2	call `error_recovery`
1	1	call `retract_last_input` return `SOMETOKEN`	call `retract_last_input` return `SOMETOKEN`
2	call `retract_last_input` return `SOMETOKEN`	2	call `retract_last_input` return `SOMETOKEN`

Then the C code above can be replaced with the following code:

```
int state = 0; /* DFA in start state */
char c;
c = nextchar();
while(1){
  state = move(state, c);
  c = nextchar();
}
```

Here the `state` variable still keeps track of the current state of the DFA, but the rest of the code is encoded directly into the transition table. The `move` function is the transition function for the DFA: given the current state and the next input character, it uses the table above to decide whether to simply move to another state (which then becomes the new value of the `state` variable) or to call the error recovery routine, or to put the last input character read back into the input and return a token.

This new C code is a bit slower than the previous C code, since every action requires consulting the transition table first, but this does not slow down the code by much. The big advantage of this version of the code is that it is much simpler than the previous version. In fact this new code is completely independent of the actual DFA it implements: all the encoding of the DFA is done in the transition table, and all the C code outside of the transition table remains the same regardless of which DFA is implemented by the table. This makes it possible to write C code which is highly optimized and very efficient, and to re-use the same code for all implementations of DFAs: only the content of the table needs to be changed, and the optimized C code needs only to be written once. This is the reason why lexers automatically generated by tools like Lex (see below) use a transition table like the new C code above. This allows the implementors of Lex to create a highly optimized version of the C code once for all, which will be used by *all* the lexers generated by Lex. All the differences between one automatically generated lexer and another will be in the content of the transition tables used by these lexers (check this yourself using the lexers you created for the homework assignments!)

Error Recovery

Now that we know how to specify tokens (sets of lexemes) using regular expressions and how to transform those regular expressions into C code (by creating NFAs from the regular expressions, then creating DFAs from the NFAs, then implementing the DFAs in C), we need to see how a lexer can recover from lexical errors when the C code that tries to detect lexemes in the input finds some input which is not matched by any regular expression.

The easy solution when a lexer finds a lexical error is to simply print an error message and terminate the compilation process. The reason we want the lexer to recover more gracefully from lexical errors is that we want the compiler to process as much of the program as possible so as to detect as many possible errors at once. If the compiler dies as soon as an error is detected, then only a single error can be detected each time the compiler is run on the input program. If the program we want to compile contains several errors we then will have to run the compiler many times before we have found and fixed all the errors in the program. Since running the compiler on the same program many times is annoying for the user, we want to create a compiler that can instead recover from errors and detect as many errors as possible at once.

When a lexical error is detected, the lexer prints an error message indicating the error to the programmer, but it is not clear what the lexer should do after that to recover from the error. The lexer knows at that point that the input is not matched by any DFA for any token in the programming language. Then the only thing the lexer can do is try to guess what the programmer really meant and try its best to change the wrong input into input that looks correct. Since the lexer cannot read the programmer's mind, the lexer can really only make guesses and try its best to recover from the error to continue processing the input. The lexer obviously cannot fix the problem in the program since the lexer cannot know for sure what the programmer really intended to do. So error recovery is not about fixing a wrong program, the program will still be wrong after error recovery happens. Error recovery is simply about trying to allow the lexer to keep processing more input in the hope that it might then be able to detect more lexical errors.

When a lexical error is detected by the lexer, the lexer has several possible solutions to try to recover:

- Ignore a character in the input.
- Insert a character in the input.
- Replace a character in the input by a new character.

- Transpose (change the order) of two adjacent characters in the input (people often type charcaters in the wrong order).

Some, none, or all of these transformations of the input might then result in input that the lexer can correctly process again. In some cases multiple transformations might be required before the lexer can recover from the error. Often there will also be multiple sequences of transformations that all lead back to correct but different inputs. In general, the best the lexer can do is to try to find the smallest sequence of transformations that changes the wrong input into correct input, and assume that that is then the right transformation to use. This is called *minimum distance error correction* (similar to the kind of network error correction codes that try to minimize the number of corrections that have to be done on some data that has gone through a network).

Some experimental compilers use such techniques to try to transform wrong input into something looking like correct input, but using such techniques is quite expensive, since the lexer has to search all the different combinations of transformations until it finds the best, smallest one. So in practice lexers often try instead to recover from errors simply by just ignoring as many characters in the input as necessary, until the input looks correct again. This is a very simplistic strategy, but in practice it is good enough to allow a lexer to recover from errors.

The Lex Lexer Generator

As you have seen in the first homework assignment, Lex (or its GNU twin called Flex) is a tool to automatically generate the C code of lexers.

The user writes in a "`.l`" specification file the regular expressions that describe the different tokens to be identified. The user also writes an action associated with each regular expression. These actions are written as C code, and are to be executed every time the corresponding regular expression matches some input.

The user then runs Lex on this specification file and from this specification file Lex produces another file `lex.yy.c` that contains C code that implements a complete lexer for these regular expressions:

$$\texttt{example.l} \;\rightarrow\; \boxed{\text{Lex}} \;\rightarrow\; \texttt{lex.yy.c}$$

What Lex does is to read the regular expressions defined in the "`.l`" file, and transform them into C code using the method we have explained above (transforming the regular expressions into NFAs, then the NFAs into DFAs, then the DFAs into C code). So Lex internally implements all the different algorithms above (like Thompson's construction method and the subset construction algorithm) for you! This is why using tools like Lex or Flex makes writing lexers much easier. In fact very few people (except students…) write lexers themselves, most people just use tools like Lex to automatically generate correct and efficient lexers from a simple specification.

The actions that were written in the specification file are copied as is in the file `lex.yy.c` at the right places (just before the C code returns a token) so that the C code of each action is executed every time the corresponding regular expression has matched some input.

The "`.l`" specification file also has two sections at the beginning and at the end of it where the user can write C declarations and C auxiliary functions that are required by the various actions associated with the regular expressions. These C declarations and auxiliary functions are then copied as is in the C code of the lexer generated by Lex. This C code should be enclosed within special braces ("%{" and "%}") in the first section, because that section can also contain regular definitions.

Once Lex has created the C code for a lexer, that lexer can then be compiled using a normal C compiler to get an executable program that implements the lexer as well as all the associated C actions that were specified in the original "`.l`" file:

$$\texttt{lex.yy.c} \quad \rightarrow \quad \boxed{\text{C Compiler}} \quad \rightarrow \quad \text{executable lexer}$$

This executable lexer can then be run by a computer and given some input. The executable lexer will then try to match this input against the regular expressions (using the C code which is the result of transforming these regular expressions into NFAs, then DFAs, then C code) and will execute the corresponding action every time a match is detected:

$$\text{input} \quad \rightarrow \quad \boxed{\text{executable lexer}} \quad \rightarrow \quad \text{tokens and output from actions}$$

Three important things need to be noted here:

- Lex itself is not a lexer, it is a *lexer generator*. Lex is a tool that you can use to create lexers, but it is not itself a lexer. Do not confuse Lex with the lexers it creates.

- Lex creates lexers that are implemented in C. This means the lexers generated by Lex can only be executed after they have been compiled by a C compiler. If you need to create a lexer that has to be implemented in Java, for example, then you cannot use Lex, you will instead have to use another tool that works very much like Lex but generates lexers that are implemented in Java. In practice there are tools like Lex available for pretty much all the programming languages you can think of. So if you want to generate automatically a lexer that is implemented in your favorite programming language, you just have to find the right lexer generator for that programming language, write a specification file for your lexer, and then apply the tool to the specification file to get the code of a lexer written in your favorite programming language.

- The programming language in which the lexer is implemented is completely unrelated to the programming language that is processed by the executable lexer as input. For example, using Lex you can only generate lexers that are written in C, but these lexers themselves can be lexers for any input language you want. If you remember you first homework assignment, the lexers you generated using Lex were all written in C (since this is what Lex does) but the lexers themselves were lexers for programming languages that contained things like "`target temperature 10`", or "`heat off`", or "`zone "foo/bar" {type hint; file "/foo/foo/"};`", etc. So do not confuse the programming language which is used to implement the lexer (which is the programming language that should then be supported by the compiler you use to compile the generated lexer) with the programming language that is processed by the executable lexer as input.

To finish this discussion of Lex, two more features of the lexers generated by Lex (these are not features of Lex itself) should be mentioned:

- As a lexer generated by Lex matches input by using DFAs implemented in C, the lexer stores the characters of the lexeme into an array of characters called `yytext`. Since the C action associated with a regular expression is executed every time the lexer has just finished matching some input with that regular expression, the C code of the action can then look at the content of the array `yytext` and it will find there the complete lexeme that was just matched by the lexer. This is useful if the lexer needs to do some extra processing on the lexeme (like storing the lexeme in the symbol table of the compiler) before the lexer returns a token to the parser.

- If a lexer generated by Lex is used in conjunction with a parser generated by Yacc (a parser generator, as we will see later) then the lexer and the parser share the global variable `yylval`. This is useful if the lexer wants to return to the parser more information than just a token (token attributes, for example): all the extra information can be store in the variable `yylval` by the lexer and the parser will then find the information there once the lexer has returned a token to it (we will see later in detail how this works from the point of the parser).

This now completes our overview of lexical analysis. The next phase in a compiler is then syntax analysis.