**Scanner Implementation 2: Regular Expressions and Finite Automata**

The other method of implementing a scanner is using regular expressions and finite automata. A quick detour for some background review and then let's see how we can generate a scanner building on techniques from automata theory.

**Regular expression review**

We assume that you are well acquainted with regular expressions and all this is old news to you.

| | |
|---|---|
| *symbol* | an abstract entity that we shall not define formally (such as "point" in geometry). Letters, digits and punctuation are examples of symbols. |
| *alphabet* | a finite set of symbols out of which we build larger structures. An alphabet is typically denoted using the Greek sigma ∑, e.g., ∑ = **{0,1}.** |
| *string* | a finite sequence of symbols from a particular alphabet juxtaposed. For example: **a**, **b**, **c**, are symbols and **abcb** is a string. |
| *empty string* | denoted ε (or sometimes ∂) is the string consisting of zero symbols. |
| *formal language* ∑* | the set of all possible strings that can be generated from a given alphabet. |
| *regular expressions* | rules that define exactly the set of words that are valid tokens in a formal language. The rules are built up from three operators: |

| | | |
|---|---|---|
| concatenation | **xy** | |
| alternation **x\|y** | **x** or **y** | |
| repetition | **x*** | **x** repeated 0 or more times |

Formally, the set of regular expressions can be defined by the following recursive rules:

1) Every symbol of ∑ is a regular expression
2) **ε** is a regular expression
3) if $\mathbf{r_1}$ and $\mathbf{r_2}$ are regular expressions, so are
   $(\mathbf{r_1})$ $\quad\mathbf{r_1r_2}$ $\quad\mathbf{r_1} \mid \mathbf{r_2}$ $\quad\mathbf{r_1}^*$
4) Nothing else is a regular expression.

Here are a few practice exercises to get you thinking about regular expressions again. Give regular expressions for the following languages over the alphabet `{a,b}`:

all strings beginning and ending in *a* _______________________
all strings with an odd number of *a's* _______________________
all strings without two consecutive *a's*_______________________

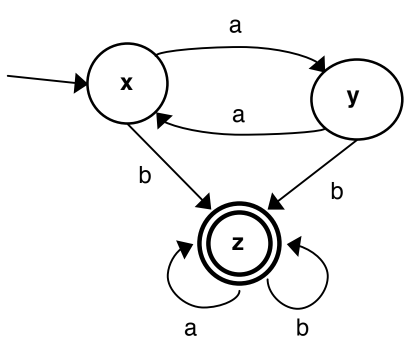
We can use regular expressions to define the tokens in a programming language. For example, here is a regular expression for an integer, which consists of one or more digits (**+** is extended regular expression syntax for 1 or more repetitions)

```
(0|1|2|3|4|5|6|7|8|9)+
```

**Finite automata review**

Once we have all our tokens defined using regular expressions, we can create a finite automaton for recognizing them. To review, a *finite automata* has:

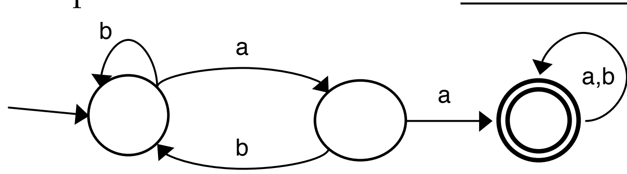
1) A finite set of states, one of which is designated the initial state or *start state*, and some (maybe none) of which are designated as final states.
2) An alphabet ∑ of possible input symbols.
3) A finite set of transitions that specifies for each state and for each symbol of the input alphabet, which state to go to next.




| | | a | b |
|---|---|---|---|
| (start) | x: | y | z |
| | y: | x | z |
| (final) | z: | z | z |

What is a regular expression for the FA above? ________________________________

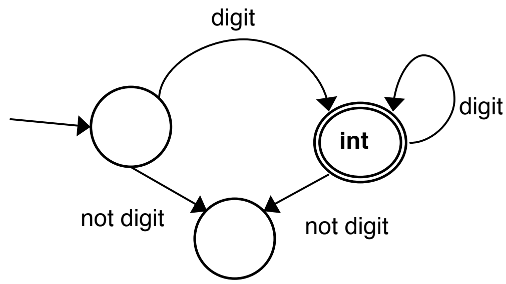
What is a regular expression for the FA below? ________________________________



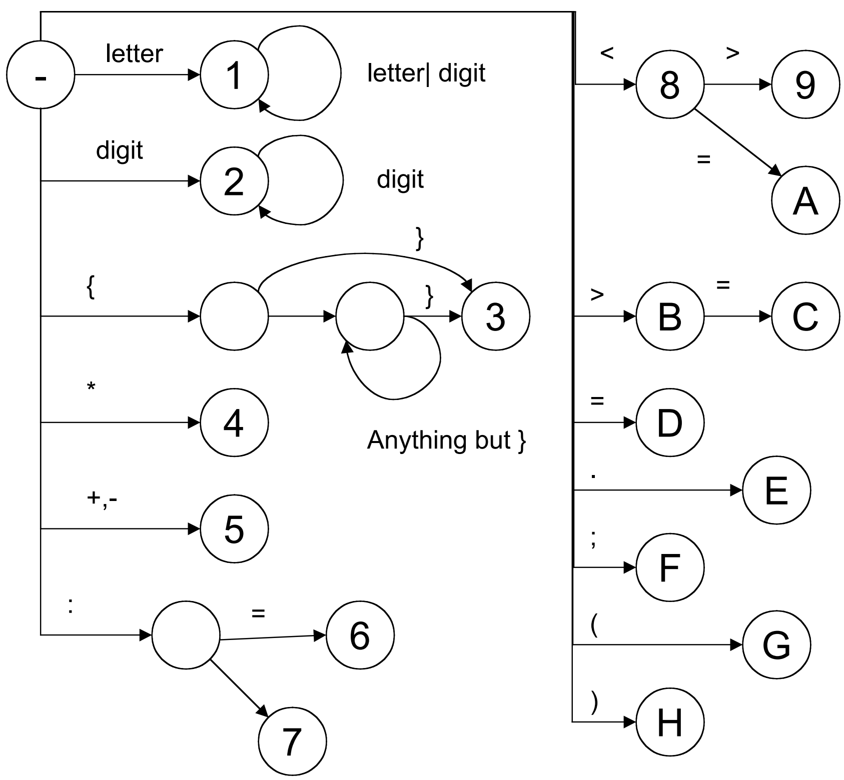

Define an FA that accepts the language of all strings that end in *b* and do not contain the substring *aa*. What is a regular expression for this language?

Now that we remember what FAs are, here is a regular expression and a simple finite automata that recognizes an integer.

(0|1|2|3|4|5|6|7|8|9)+




Here is an FA that recognizes a subset of tokens in the Pascal language:




This FA handles only a subset of all Pascal tokens but it should give you an idea of how an FA can be used to drive a scanner. The numbered/lettered states are final states. The loops on states 1 and 2 continue to execute until a character other than a letter or digit is read. For example, when scanning "**`temp := temp + 1;`**" it would report the first token at final state 1 after reading the "**`:`**" having recognized the lexeme "**`temp`**" as an identifier token.

What happens in an FA-driven scanner is we read the source program one character at a time beginning with the start state. As we read each character, we move from our current state to the next by following the appropriate transition for that. When we end up in a final state, we perform an action associated with that final state. For example, the action associated with state 1 is to first check if the token is a reserved word by looking it up in the reserved word list. If it is, the reserved word is passed to the token stream being generated as output. If it is not a reserved word, it is an identifier so a procedure is called to check if the name is in the symbol table. If it is not there, it is inserted into the table.

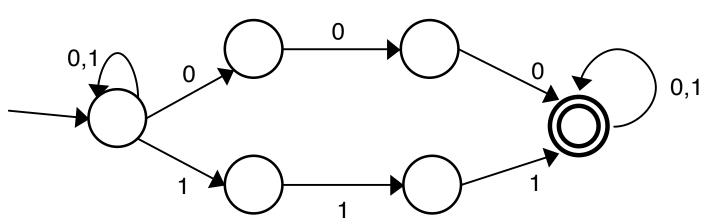
Once a final state is reached and the associated action is performed, we pick up where we left off at the first character of the next token and begin again at the start state. If we do not end in a final state or encounter an unexpected symbol while in any state, we have an error condition. For example, if you run "ASC@I" through the above FA, we would error out of state 1.

**From regular expressions to NFA**

So that's how FAs can be used to implement scanners. Now we need to look at how to create an FA given the regular expressions for our tokens. There is a looser definition of an FA that is especially useful to us in this process. A *nondeterministic finite automaton (NFA)* has:

1) A finite set of states with one start state and some (maybe none) final state
2) An alphabet ∑ of possible input symbols.
3) A finite set of transitions that describe how to proceed from one state to another along edges labeled with symbols from the alphabet (but not ε). We allow the possibility of more than one edge with the same label from any state, and some states for which certain input letters have no edge.

Here is an NFA that accepts the language **`(0|1)*(000|111)(0|1)*`**




Notice that there is more than one path through the machine for a given string. For example, **`000`** can take you to a final state, or it can leave you in the start state. This is

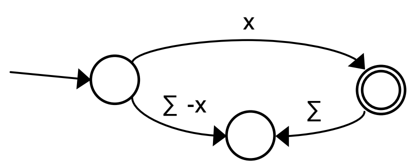
where the non-determinism (choice) comes in. If any of the possible paths for a string leads to a final state, that string is in the language of this automaton.
There is a third type of finite automata called ε-NFA which have transitions labeled with the empty string. The interpretation for such transitions is one can travel over an empty-string transition without using any input symbols.

A famous proof in formal language theory (Kleene's Theorem) shows that FAs are equivalent to NFAs which are equivalent to ε-NFAs. And, all these types of FAs are equivalent in language-generating power to that of regular expressions. In other words,

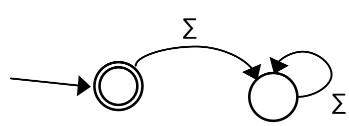
> If R is a regular expression, and L is the language corresponding to R, then there is an FA that recognizes L. Conversely, if M is an FA recognizing a language L, there is a regular expression R corresponding to L.

It is quite easy to take a regular expression and convert it to an equivalent NFA or ε-NFA, thanks to the simple rules of Thompson's construction:

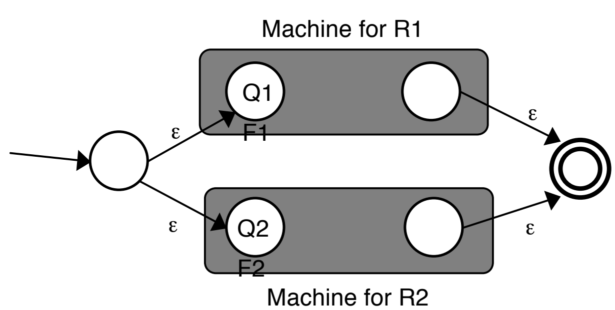
*Rule 1*: There is an NFA that accepts any particular symbol of the alphabet:



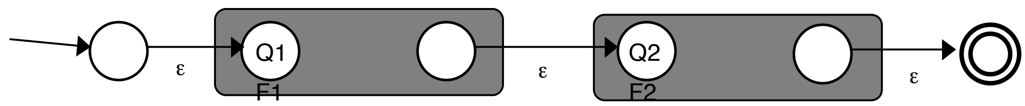

*Rule 2*: There is an NFA that accepts only ε:



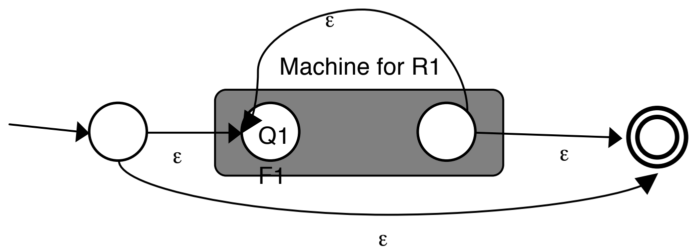

*Rule 3*: There is an ε-NFA that accepts `r1|r2`:




*Rule 4*: There is an ε-NFA that accepts `r1r2`:




*Rule 5*: There is an ε-NFA that accepts `r1*`:

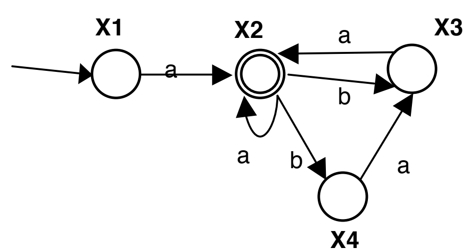



**Going Deterministic: Subset Construction**

Using Thompson's construction, we can build an NFA from a regular expression, we can then employ *subset construction* to convert the NFA to a DFA. Subset construction is an algorithm for constructing the deterministic FA that recognizes the same language as the original nondeterministic FA. Each state in the new DFA is made up of a set of states from the original NFA. The start state of the DFA will be the start state of NFA. The alphabet for both automata is the same.

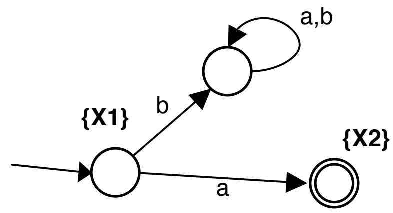
So, given a state of from the original NFA, an input symbol x takes us from this state to the union of original states that we can get to on that symbol x. We then have to analyze this new state with its definition of the original states, for each possible input symbol, building a new state in the DFA. The states of the DFA are all subsets of S, which is the set of original sets. There will be a max of $2^n$ of these (because we might need to explore the entire power set), but there are usually far fewer. The final states of the DFA are those sets that contain a final state of the original NFA.

Here is an example:

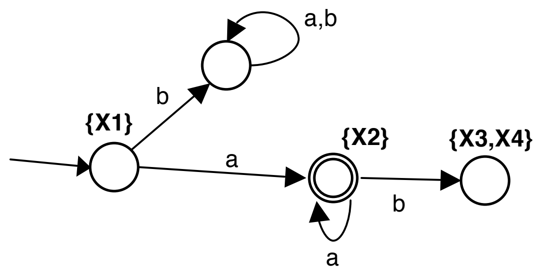



This is non-deterministic for several reasons. For example, the two transitions on ‘b’ coming out of the final state, and no ‘b’ transition coming out of the start state. To create an equivalent deterministic FA, we begin by creating a start state, and analyzing where we go from the start state in the NFA, on all the symbols of the alphabet. We create a set of states where applicable (or a sink hole if there were no such transitions). Notice if a final state is in the set of states, then that state in the DFA becomes a final state.

(after creating start state and filling in its transitions)

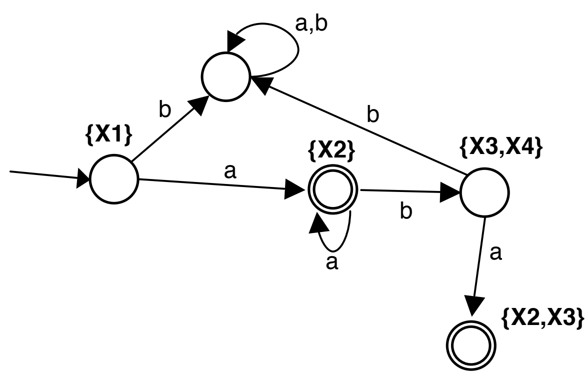



(after filling in transitions from {X2} state)

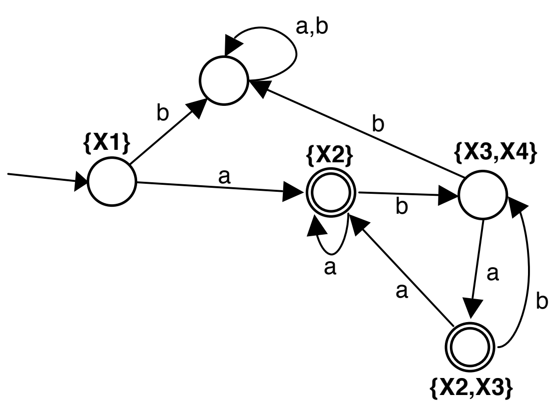



We continue with this process analyzing all the new states that we create. We need to determine where we go in the NFA from each state, on all the symbols of the alphabet.

(after filling in transitions from {X3, X4} state)




And finally, filling in the transitions from {X2, X3} state brings us full circle. This is now a deterministic FA that accepts the same language as the original NFA. We have 5 states instead of original 4, a rather modest increase in this case.



The process then goes like this: from a regular expression for a token, we construct an NFA that recognizes them using Thompson's algorithm. NFAs are not useful as drivers for programs because non-determinism implies choices and thus, expensive exhaustive backtracking algorithms. So, we use subset construction to convert that NFA to a DFA. Once we have the DFA, we can use it as the basis for an efficient non-backtracking scanner.

### `lex`: a scanner generator

The reason we have spent so much time looking at how to go from regular expressions to finite automata is because this is exactly the process that **`lex`** goes through in creating a scanner. **`lex`** is a lexical analysis generator that takes as input a series of regular expressions and builds a finite automaton and a driver program for it in C through the mechanical steps shown above. Theory in practice!

### Programming Language Case Study: FORTRAN I

The first version of FORTRAN is interesting for us for a couple reasons. First of all, the language itself violates just about every principle of good design that we specified in a previous handout. Secondly, the process of developing the first FORTRAN compiler laid the foundation for the development of modern compilers.

Here is a brief description of FORTRAN I:

- Variables can be up to five characters long and must begin with a letter. Variables beginning with **`i`**, **`j`**, **`k`**, **`l`**, **`m`** and **`n`** are assumed by the compiler to be integer types, all others are reals. (This restriction may very well be the reason why programmers continue to use **`i`** as the name of the integer loop counter variable…)

- Variable declarations are not required, except for arrays. Arrays are limited to 3 dimensions.

- Computations are done with the assignment statement where the left side is a variable and the right-side is an equation of un-mixed types.

- The control structures consisted of:

    - unconditional branch: **`GOTO`** `<statement number>`

    - conditional branch: **`GOTO (30,40,50) I1`** where **`I1`** has a value of **`1`**, **`2`** or **`3`** to indicate which position in the list to jump to.

    - **`IF`** statement: **`IF (A+B-1.2) 7, 6, 9`** transfers control to statement labeled **`7`**, **`6`**, or **`9`** depending on whether **`(A+B-1.2)`** is negative, zero or positive.

  - **`DO`** statement: **`DO 17 I = 1, N, 2`** which specifies that the set of statements following, up to and including that labeled **`17`**, is to be executed with **`I`** first assigned value **`1`**, incrementing by **`2`**, up through **`N`**.

- No user-defined subroutines allowed, although several built-ins were provided including **`READ`** and **`WRITE`** for I/O.

- Blanks are ignored in all statements, e.g., all the following are equivalent:

  - **`DIMENSION IN DATA(10000)`**
  - **`DIMENSIONINDATA(10000)`**
  - **`D I M E N S I O N I N D A T A ( 1 0 0 0 0 )`**

- Reserved words are valid variable names, e.g., **`DIMENSION IF(100)`** declares an array called **`IF`**.

The task of building the first compiler was a huge one compared to what we face today. First of all, we can see there were considerable challenges in the definition of the language itself. Allowing variables to have the same name as reserved words makes the scanning process much more difficult (although they didn't really do "scanning" in the first compiler). Not requiring variable declarations meant the compiler had to remember lots of rules, and try to infer data type from context. In addition, these early compiler-writers had no formalisms to help as we do today, e.g., regular expressions and context-free grammars. Everything had to be designed and written from scratch. These formal techniques and the automatic tools based on them, allow us to build a fairly involved compiler over the course of 8 weeks, which is a far cry from 18 person-years.

The first FORTRAN compiler itself started as three phase but by the end, it had expanded to six. The first phase read the entire source program, filing all relevant information into tables, and translating the arithmetic expressions into IBM 704 machine code. The second phase analyzed the entire structure of the program in order to generate optimal code for the **`DO`** statements and array references. Phase 3 was the code generator as we know it. After a certain amount of work on phase 2, additional phase were added (3, 4, 5 with 6 becoming the code generator) to do further optimizations. Recall that optimization was key to the acceptance of the compiler since early FORTRAN programmers were skeptical of a machine creating truly optimal assembly code.

Comparing this structure to our "modern" compiler structure, we see that it does not map too well. In the original FORTRAN compiler, expressions are scanned, parsed and code-generated in the first and immensely complicated kitchen-sink of a first phase. One could view almost all of their inner four phases as just one great big optimization phase. So this compiler looks very different from what we know. Still, the same tasks were performed although in some cases less efficiently than what we can do today. The most important achievement, besides creating the first working compiler (which was a considerable achievement), was the foundations laid in optimization techniques. We will see that many of these techniques are still used in modern compilers today.

**Bibliography**

A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. Reading, MA: Addison-Wesley, 1986.

J. Backus, "The History of FORTRAN I, II, III.", SIGPLAN Notices, Vol. 13, no. 8, August, 1978, pp. 165-180.

J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

D. Cohen, Introduction to Computer Theory, New York: Wiley, 1986.

C. Fischer, R. LeBlanc, Crafting a Compiler. Menlo Park, CA: Benjamin/Cummings, 1988.

J. Hopcroft, J. Ullman, Introduction to Automata Theory, Languages, and Computation, Reading MA: Addison-Wesley, 1979.

S. Kleene, "Representation of Events in Nerve Nets and Finite Automata," in C. Shannon and J. McCarthy (eds), Automata Studies, Princeton, NJ: Princeton University Press, 1956.

A. McGettrick, The Definition of Programming Languages. Cambridge: Cambridge University Press, 1980.

T. Sudkamp, Languages and Machines: An Introduction to the Theory of Computer Science, Reading, MA: Addison-Wesley, 1988.

R.L.Wexelblat, History of Programming Languages. London: Academic Press, 1981.