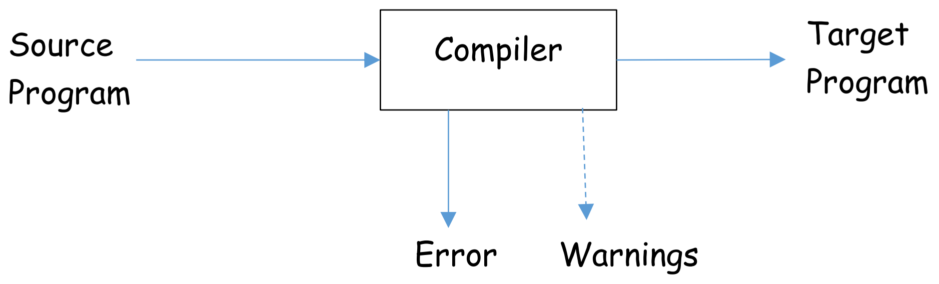
Introduction

A **compiler** is a *system software that converts a program written in high-level language that is suitable for programmers into a low-level language required by computers*. During the process, the compiler will also attempt to spot and report obvious programmer mistakes. This is illustrated below:



There are two parts of compilation:

i). *The analysis part* – breaks up the source program into constant pieces and creates an intermediate representation of the source program.

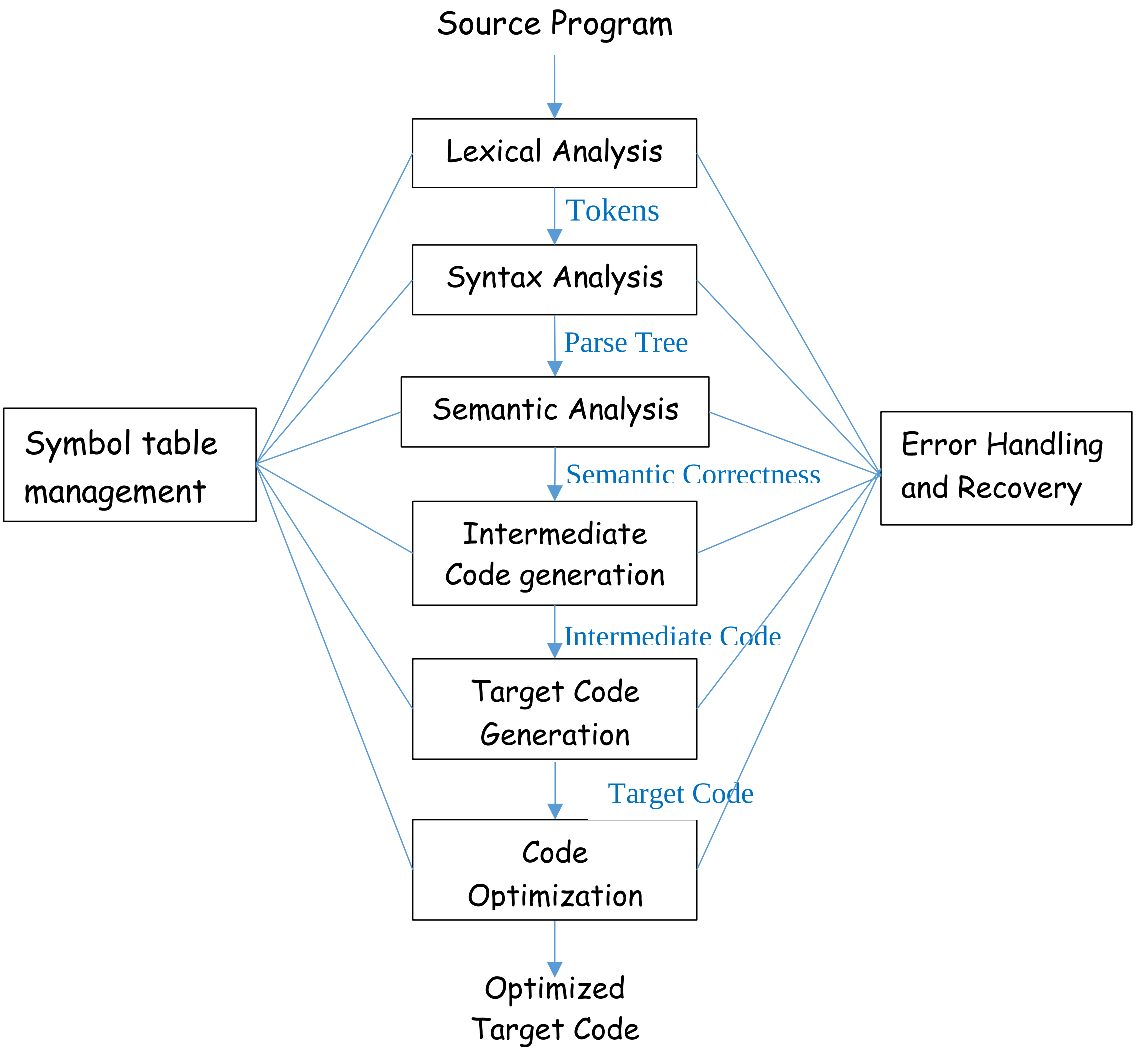
ii). *Synthesis part* – constructs desired target program from the intermediate representation.

Phases of a Compiler

To ease the process of development and understanding, a compiler can be conceptually divided into the following phases:

i) Lexical analysis
ii) Syntax analysis
iii) Semantic analysis
iv) Intermediate generation
v) Target code generation
vi) Code optimization
vii) Symbol table management
viii) Error handling and recovery

The following diagram shows the relationship between these modules:



i). Lexical Analysis

This is the initial part of reading and analysing the program text; the text is read and **divided** into ***tokens***, each of which *corresponds to a symbol* in the programming language e.g. a variable name, number, keyword, etc. It is basically used to identify valid words in the input source program.

ii). Syntax Analysis

This phase takes the list of tokens produced by the lexical analysis and arranges them into a tree structure (syntax tree) that reflects the structure of the program. It is generally used to establish if the program is grammatically correct. This phase is often called **parsing**.

iii). Semantic Analysis

This phase analyses the syntax tree to determine if the program violates certain consistency requirements e.g. if a variable is used and not declared or if it is used in a context that doesn't make sense given the type of variable such as trying to assign a value greater than the variable.

iv). Intermediate Code Generation

This phase is used to translate the program into a simple machine independent intermediate language. *This process helps to retarget the compiler generating code from one processor to another*.

Generally, generating machine code directly from source code entails two problems

- With m languages and n target machines, we need to write m × n compilers
- The code optimizer which is one of the largest and very-difficult-to-write components of any compiler cannot be reused

By converting source code to an intermediate code, a machine-independent code optimizer may be written

v). Target Code Generation

This is used for template substitution i.e. for every statement of the intermediate language, a predefined target language template is used to generate the final code. Basically, it is used to translate the intermediate language to specific machine architecture.

vi). Code Optimization

This phase is used to make the code efficient. It is generally used to *eliminate redundant codes generated during the process of compilation.*

vii). Symbol Table Management

A **symbol table** is a data structure that holds information about all symbols held in the source program. Information stored includes variable names, their size, types, etc. it does not form part of the final code generated.

viii). Error Handling and Recovery

Error handling is used to give an indication regarding the type of error. It should be detailed enough to pin-point the error and at the same time should not be too wordy to confuse the user.

Recovery is used to undo some of the processing already carried by the parser so as to reach a state where it can proceed again i.e. prevents parser from coming out on the first error it encounters.

Qualities of a Good Compiler

i). **Correct**: the meaning of sentences must be preserved

ii). **Robust**: wrong input is the common case

- Compilers and interpreters can’t just crash on wrong input
- They need to diagnose all kinds of errors safely and reliably

iii). **Efficient**: resource usage should be minimal in two ways

- The process of compilation or interpretation itself is efficient
- The generated code is efficient when interpreted

iv). **Usable**: integrate with environment, accurate feedback

- Work well with other tools (editors, linkers, debuggers, . . .)
- Descriptive error messages, relating accurately to source

Example of a Compilation Process

Consider the following program:

```
program
     var x1,x2: integer; {integer variable declaration}
     var xR: real;       {real variable declaration}
begin
     xR:=x1+x2*10;       {assignment statement}
end.
```

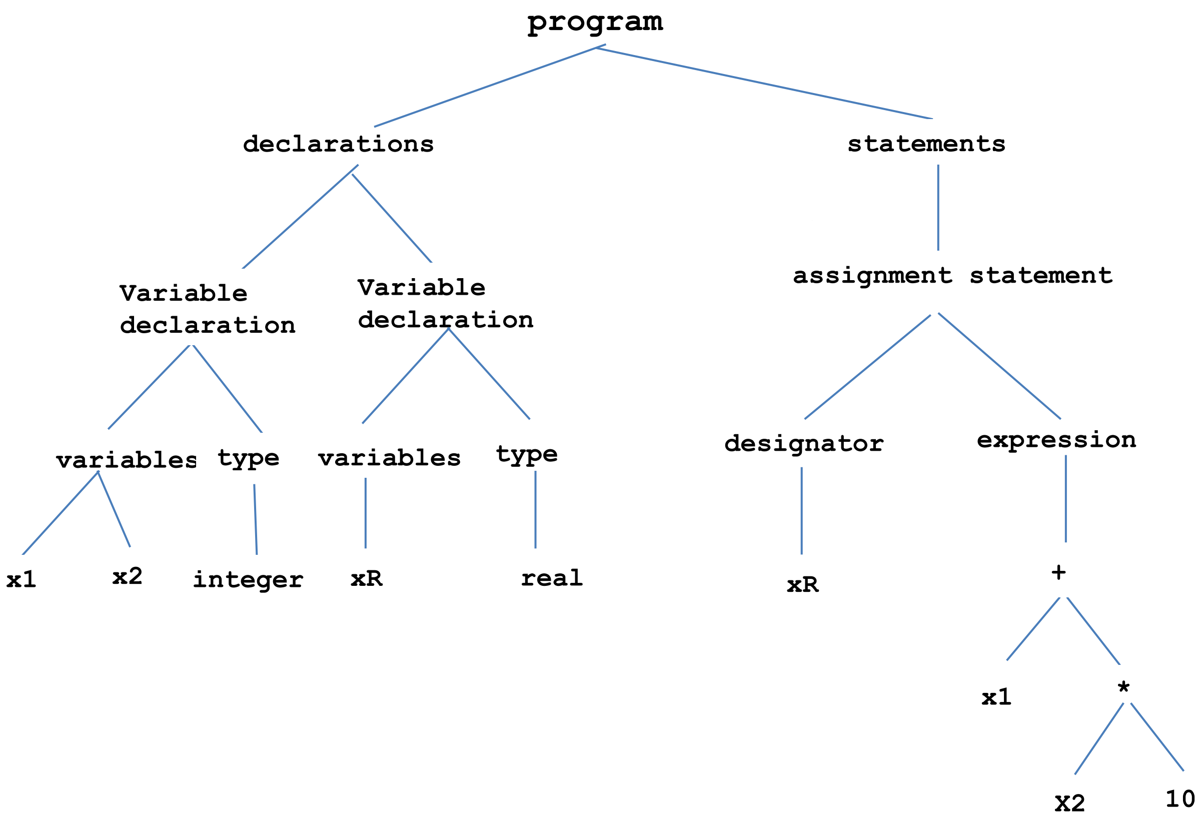
1. Lexical Analysis

The lexical analyser scans the program looking for major lexical elements (tokens). It produces the following:

```
program,var,x1,’,’,x2,’:’,integer,’;’,var,xR,’:’,real,’;’,begin,xR,
’=’,x1,’+’,x2,’*’,10,’;’,end,’.’
```

2. Syntax Analysis

It analyses for grammatical correctness and for a grammatically correct program. It produces the following parse tree:



3. Code Generation

It transverses the parse tree and generates code for the input program. Basically, it assumes a stack oriented machine with instructions such as PUSH, ADD, MULT, STORE. The following is a sample code for the program:

```
PUSH x2
PUSH 10
MULT
PUSH x1
ADD
PUSH @ xR
STORE
```

Note: @returns the address of the variable following it.

4. Symbol Table

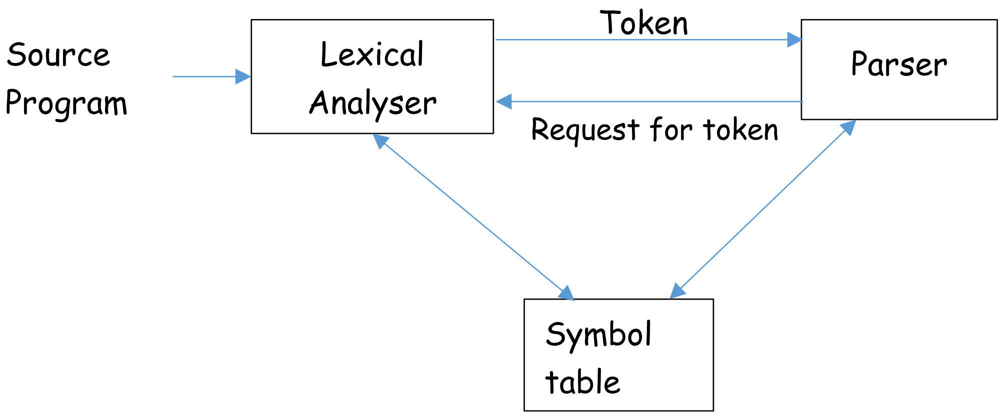
It is a table created by the lexer and parser and later used by the code generator. The following is the table generated.

Name	Class	Type
x1	Variable	Integer
x2	Variable	Integer
xR	Variable	Real

Lexical Analysis

The word “lexical” in traditional sense, means “*pertaining to words*”. In terms of programing languages, words are objects like variable names, numbers, keywords etc. such words are traditionally called tokens.

A lexical analyser (lexer) will take as its **input** a string of individual words and divide them into tokens. Additionally, it will filter out layout-out characters (spaces, new lines etc.) and comments. Apart from this, the lexical analyser also participates in creation and maintenance of a symbol table as shown below:



Lexical analysis specifications are traditionally written using **regular expressions** – *An algebraic notation for describing sets of strings OR a notation used to specify patterns corresponding to a token*.

Once the set of regular expressions are ready, a **finite automata** is constructed to determine whether or not a given word belongs to the language.

Definitions:

1. **Token** – a lexical token is a sequence of ***characters*** that can be treated as a **unit** in the **grammar** of programing language e.g. a type token can consist of an identifier, number etc. examples of non-tokens includes comments, macros, pre-processor directives, tabs, new lines, etc.
2. **Patterns** – this is a set of ***strings*** for which the same token is produced as output. Regular expressions are an important notation for specifying patterns e.g. the pattern for Pascal identifier token id is $id \to letter(letter|digit)^*$.
3. **Lexeme** – this is a sequence of ***characters*** in the source program that is ***matched*** by the pattern for a token e.g. the lexer will return the same token to the parser whenever it comes across the pattern that contains the following six lexemes: (=,<>,<,<=,>,>=)

Specifications of tokens

The set of all variable names are sets of strings, where the individual letters are taken from a particular alphabet. Such a set of strings is called a language. Therefore, a **language** is a set of finite length sequences of elements drawn from a specified finite set A of symbols. It is analogous to a collection of words.

In this case, the set A is called the alphabet of L, whose elements are called words. An alphabet might be $\{a, b\}$, and a string of that alphabet might be $ababba$. The empty word (length zero string) is allowed and is often denoted by $e, \varepsilon \ or \ \Lambda$.

Given an alphabet, it is possible to describe a set of strings using a regular expression i.e. regular expressions of a language are created by combining its alphabet.

For a language, L, with alphabet set $\sum$, the following rules define regular expressions:

i) e is a regular expression denoting the language $\{e\}$ i.e. the set containing only the empty string.
ii) If ‘a’ is a special symbol in $\sum$ then ‘a’ denotes a regular expression corresponding to the language $\{a\}$ i.e. set containing only string ‘a’.
iii) If r_1 and r_2 are regular expressions corresponding to the languages L_1 and L_2 respectively then:

1. $r_1|r_2$ is a regular expression corresponding to the language $L_1 \cup L_2$ i.e. set containing all strings in : L_1 and L_2
2. r_1r_2 is a regular expression corresponding to the language created by concatenating strings of L_2 to strings of L_1
3. r_1^* is a regular expression corresponding to the language L_1^* i.e. the set containing zero or more occurrences of the strings belonging to L_1
4. (r_1) is a regular expression corresponding to the language L_1 itself.

Note: the unary operator * has the highest precedence followed by concatenation while alteration '|' has the lowest precedence.

Examples
Consider the following regular expressions for a simple alphabet consisting of only two letters: $\Sigma = \{0,1\}$

ix) $(0|1)^*$ will denote all binary strings including the empty string
x) $(0|1)(0|1)^*$ will denote all non-empty binary strings.
xi) $0(0|1)^*0$ denotes all strings of length of at least two, starting and ending with zeros.
xii) $(0|1)^*0(0|1)(0|1)$ denotes all the binary with at least three characters, where the third last character is always zero.
xiii) $0^*10^*10^*10^*$ denotes all binary strings possessing exactly three ones.

Most programming languages define variables as $letter(letter|digit|'\_‘)^*$ i.e. a variable name is a sequence of letters, digits and the '_" but the first character must be a letter. The set of signed integers may be specified as:

$(+|e|-)digit(digit)^*$ OR $(+|-|e)[0-9]([0-9])^*$

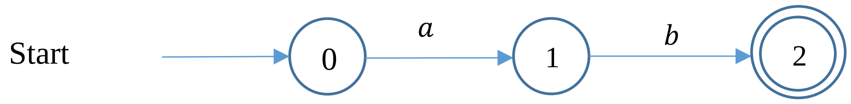
A set of floating point numbers may be defined as: ______________________________

Exercise
i) Write down the regular expression for the following:
   a) Binary strings such as '0' followed by '1'
   b) To check an IP address for correct syntax, four groups of 1 - 3 digits separated by periods
   c) Any decimal number that is a multiple of 5
ii) Describe the following regular expressions
   a) $a^*(a|b)$
   b) $(a|b)^*abb$

Finite Automata

This is a recognizer which is used to identify the tokens occurring in an input stream. Basically, it is a machine with a finite number of states and a finite number of transitions between them.

There is a distinctive start state where the machine starts. Between the states, there are transitions *labelled by the alphabets of the language*, for instance, if the machine is at state S_i and there is a transition labelled by alphabet $a_k \in \Sigma$ to the state S_j then the machine can perform such a transition provided that the current symbol is a_k. Starting from the start state, the machine finally reaches some state after exhausting all the input symbols. The **final** state reachable **for all the input streams** (**tokens**) is called the ***acceptor/final state*** e.g. the following is a finite automata for the string ab.



If the machine is in the state S_i and there does not exist any possible transition from S_i to the next input symbol then the input is rejected.

There are two types of finite automata:

i) Non-deterministic Finite Automata (NFA)
ii) Deterministic Finite Automata (DFA)

I). Non-deterministic Finite Automata (NFA)

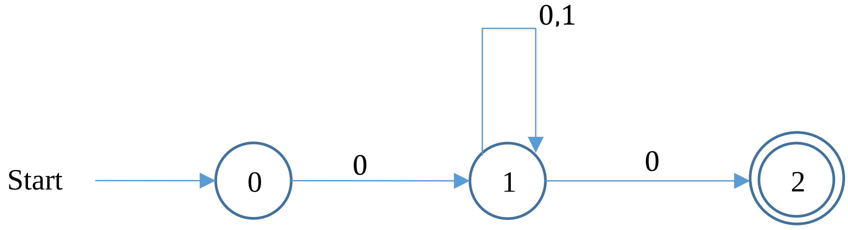
This is a finite automata that has ability of **more than one possible transitions** from a state on the same input symbol. The actual transition occurring is selected non-deterministically by the machine.

Another non-determinism lies within the empty transitions (e-transition). If there is an **e-transition** from state S_i to state S_j then the machine can do such a transition without consuming any other input.

An input is said to be accepted/recognized by the automata if **there exists at least one path from the start state of the machine to the final state** whose transitions are governed by the input stream.

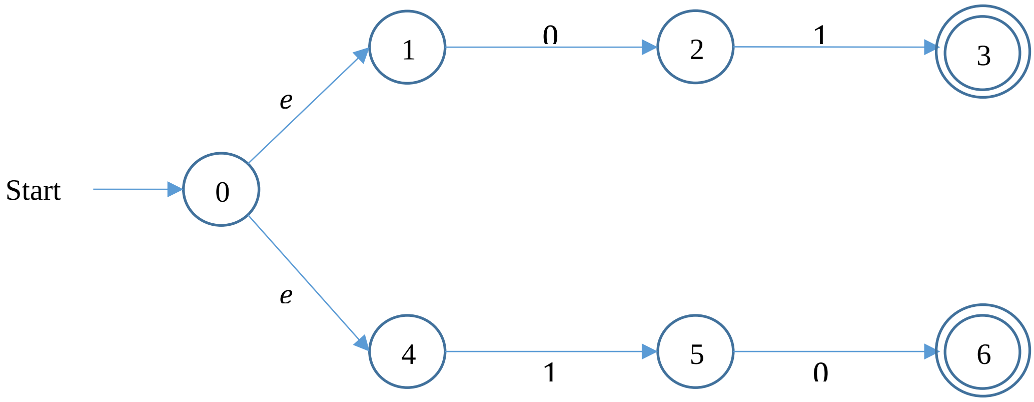
Examples

i) Construct an NFA for the regular expression $0(0|1)^*0$



Note: from state 1, there are two transitions labelled by 0, one to state 1 itself and the other to 2state one itself and the other to state two.

ii) Construct NFA for the regular expression 01|10



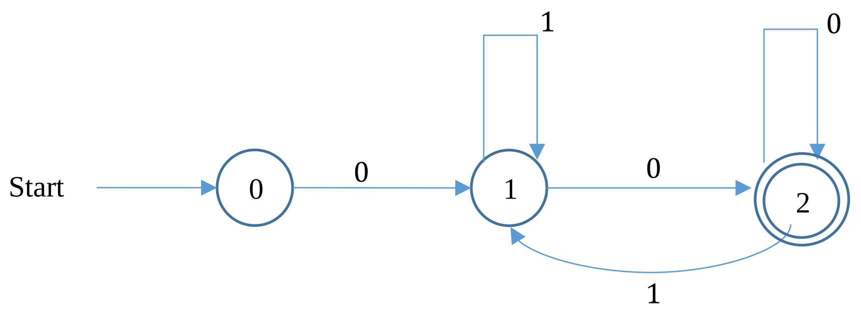
II). Deterministic Finite Automata (DFA)

This is a finite automata that cannot have **more than one transition emanating** from the same state labelled by the same input symbol. Also, there cannot be any **empty transitions**.

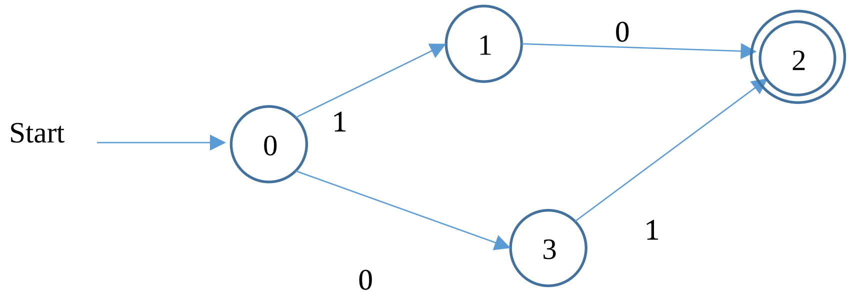
Examples

Construct a DFA for the following regular expressions:

i) $0(0|1)^*0$



ii) 01|10



Note: for each and every non-deterministic alternative of an NFA gets converted into an explicit path from start to final state in a DFA. This often results in an explosion in the number of states in a DFA.

The following is an algorithm to test whether an input stream is accepted by a DFA or not:

Algorithm: DFA-test

Input: a DFA with a start state S_0.

An input stream

Output: “yes” is accepted, “no” otherwise.

```
Begin
    S = S0
    While not end − of − input do
        let c = next input symbol
        if there is a transition on c from S to S1 then
            S = S1
    end while
    if S is a final state and c = end − of − input then
        return "yes"
    else
        return "no"
```

Note: the existence of non-determinism makes the task of testing if an input stream is acceptable difficult since it requires trying all possible alternatives that may occur in non-deterministic transition of states.

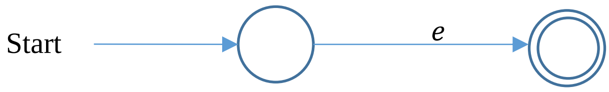
Regular Expression to NFA

The conversion of a regular expression to NFA consists of breaking down the regular expression into the simplest sub-expressions, constructing the corresponding NFAs, and then combining these small NFAs guided by the operations of the regular expression.

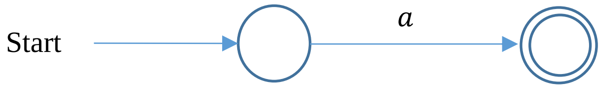
This construction is known as Thompson's construction rules.

The following are Thompson's construction rules:

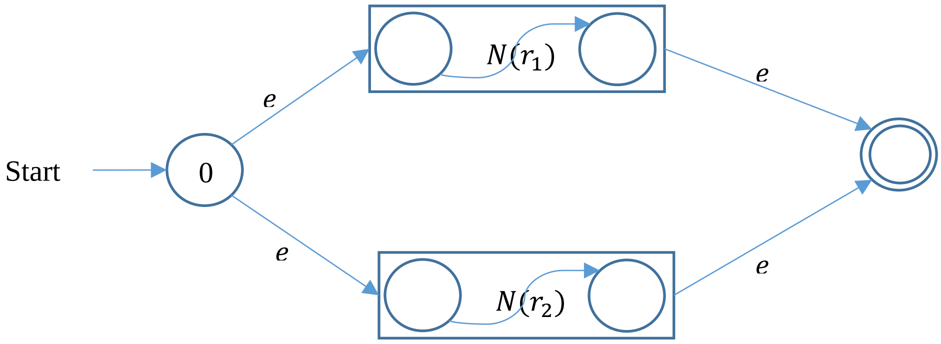
i) For e, the NFA consists of two states (start and final state). The transition is labelled by (e).



ii) For any alphabet symbol 'a' in the alphabet set Σ, the NFA also consists of two states with transition labelled by a.

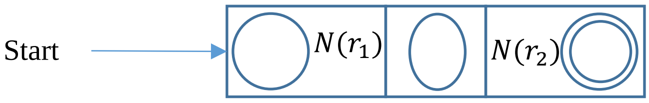


iii) For regular expression $\mathrm{r_1|r_2}$, if $\mathrm{N(r_1)}$ is NFA for $\mathrm{r_1}$ and $\mathrm{N(r_2)}$ is NFA of $\mathrm{r_2}$ then NFA of $\mathrm{N(r_1|r_2)}$ is constructed as follows:

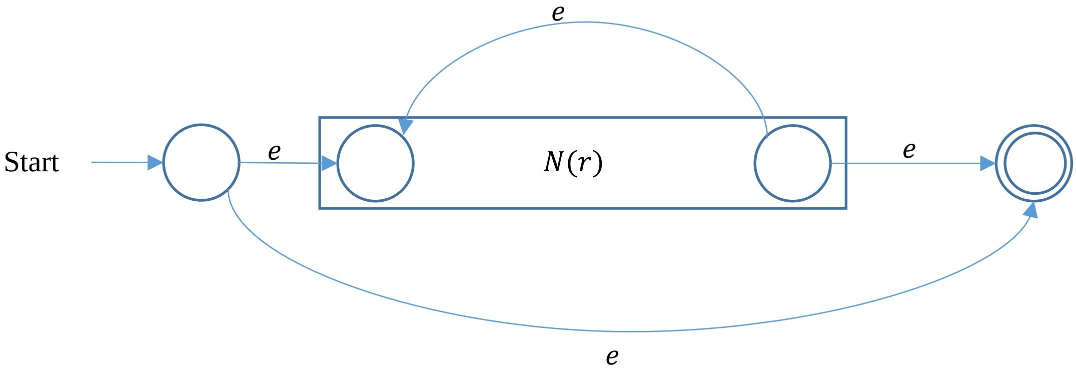


i.e. e-transitions are introduced from the new start state to the start states of $N(r_1)$ and $N(r_2)$ and similarly from the final state of $N(r_1)$ and $N(r_2)$ to the newly created final state.

iv) For the regular expression r_1r_2, the NFA $N(r_1r_2)$ is constructed by merging the NFAs $N(r_1)$ and $N(r_2)$ i.e. the final state of $N(r_1)$ is merged with the start state of $N(r_2)$ as shown:



v) For regular expression r^*, $N(r^*)$ is constructed as follows:



i.e. e-transitions from the new start state to the new final state correspond to zero occurrences of r, whereas, from the final state to the initial state of $N(r)$ corresponds to the repeated occurrence of r

vi) If $N(r)$ it is NFA for a regular expression, it is also NFA for the parenthesized expression (r).

Example

Use Thompson construction rules to come up with the NFA of the following regular expression

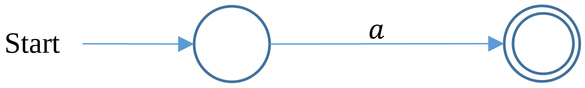
$$a(a|b)^*ab$$

The sub expressions are:

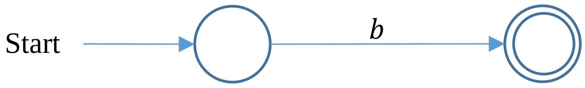
1. a
2. b
3. $a|b$
4. $(a|b)^*$
5. $a(a|b)^*$
6. ab
7. $a(a|b)^*ab$

The corresponding NFA's are as shown below:

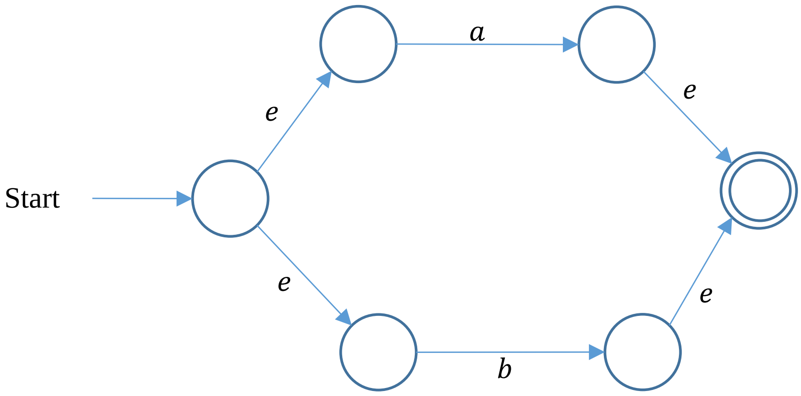
1. a



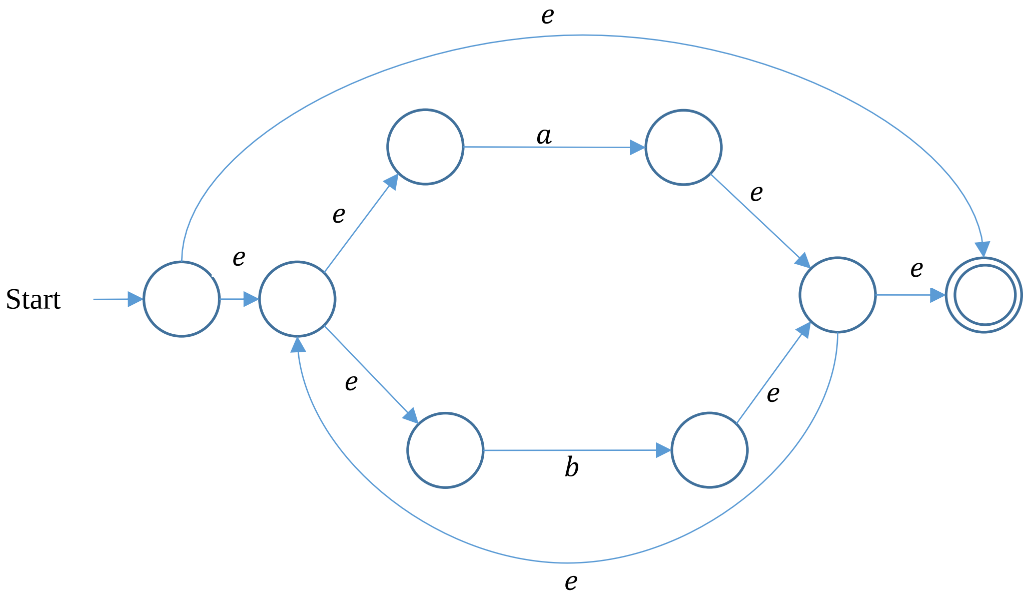
2. b



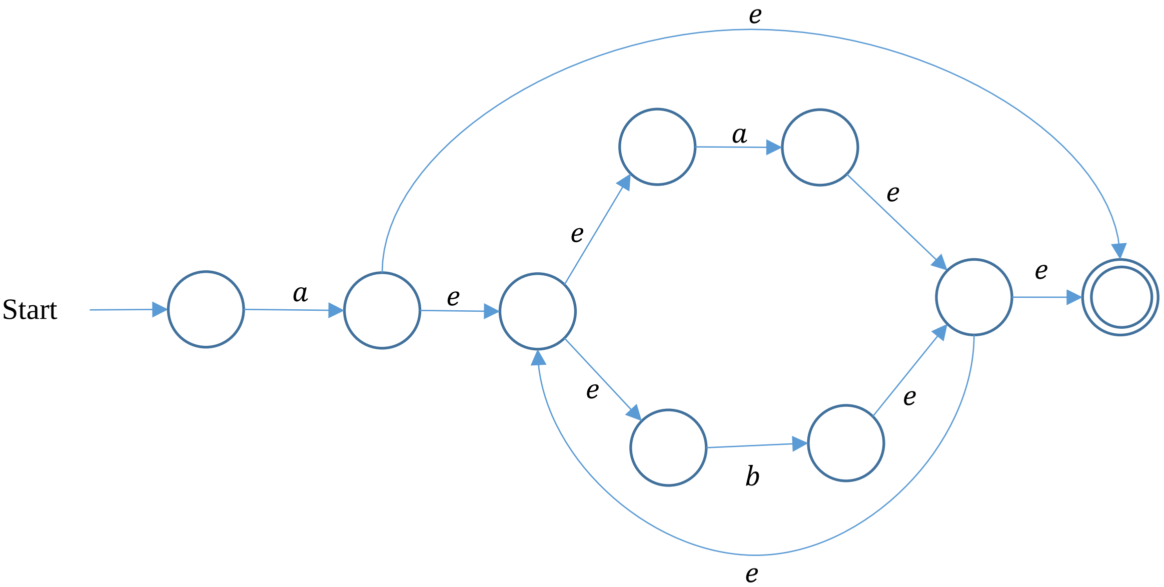
3. $a|b$



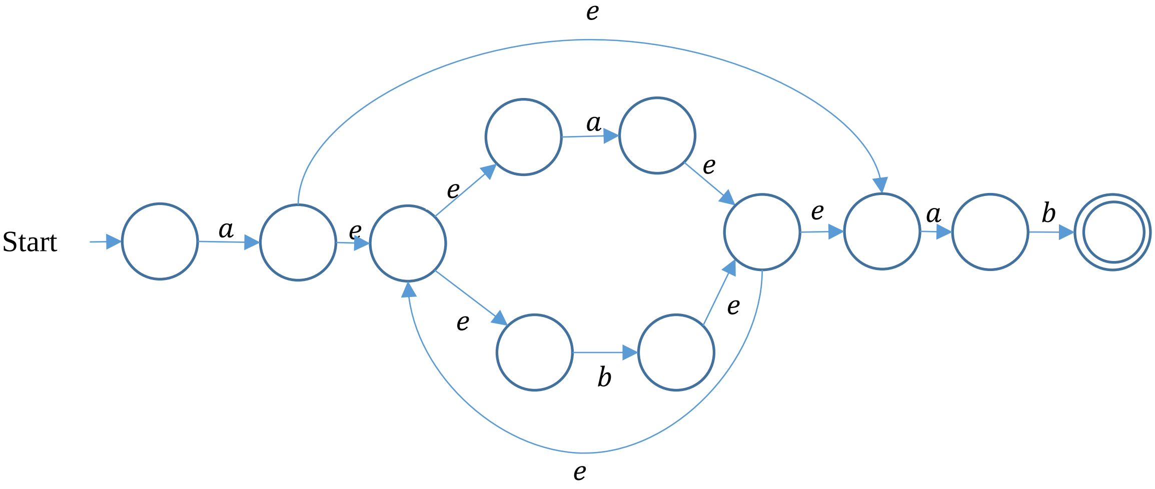
4. $(a|b)^*$



5. $a(a|b)^*$



6. $a(a|b)^*ab$



Exercise

1. Construct NFAs for the following regular expressions:
   a. $a(a|b)^*a$
   b. $(a|b)^*abb$
   c. $(a|b)^*a(a|b)(a|b)(a|b)$
   d. $a^*(a|b)$
2. Apply Thompson's rules to create NFA for the regular expression
   $((0|1)^*00)|0$