NFA to DFA Converter

A Project by

Tricia Jennifer F. Abella
Enrique Ma. R. Sarmiento

Submitted to

Luisito Agustin
Instructor, CE 160

In Partial Fulfillment of the Requirements for the Course
CE 160: Automata and Formal Languages

Department of Electronics, Computer and Communications Engineering
School of Science and Engineering
Loyola Schools
Ateneo de Manila University
Quezon City, Philippines

October 2004

Abstract

Every language that can be described by some NFA can also be described by some DFA. That is to say, every NFA has an equivalent DFA. This project converts an NFA, with no epsilon transitions, to an equivalent DFA.

The program was written in C++ and compiled with Dev-C++. It is console-based, implements dynamic memory allocation, and uses the complete subset construction for the conversion.

Acknowledgments

We thank the following for helping us come up with this magnum opus:

- Our blockmates, for being one with us in the agony.
- Chris San Buenventura, for helping us split the main function into smaller functions.
- Sir Lui, for challenging us enough.
- And God, for bringing us together.

Table of Contents

1. Introduction

1.1. Theoretical Background

Finite automata can be deterministic or nondeterministic. These two kinds of automata differ in their transitions. On each input, a deterministic finite automaton or DFA allows one and only one state to which it can transition from its current state. Nondeterministic automata or NFA, on the other hand, can be in several states at once.

Although there are many languages for which an NFA is easier to construct than a DFA, it is a surprising fact that every language that can be described by some NFA can also be described by some DFA. Moreover, the DFA in practice has about as many states as the NFA, although it often has more transitions. In the worst case, however, the smallest DFA can have 2^n states while smallest NFA for the same language has only *n* states. The proof that DFA's can do whatever NFA's can do involves an important "construction" called the *subset construction* because it involves constructing all subsets of the set of states of the NFA (Hopcroft et al, 2001).

The subset construction starts from an NFA $N = (Q_N, \sum, \delta_N, q0, F_N)$. Its goal is the description of a DFA $D = (Q_D, \sum, \delta_D, \{q0\}, F_D)$ such that $L(D) = L(N)$. The input alphabets ($\sum$) of the two automata are the same and the start state of *D* is the set containing only the start state of *N (q0)*. Q_D is the set of subsets of Q_N; that is, Q_D is the *power set* of Q_N. If Q_N has *n* states, then Q_D will have 2^n states. Often, not all these states are accessible from the start state Q_D. Inaccessible states can be "thrown away," so effectively, the number of states of *D* may be much smaller than 2^n. F_D is the set of subsets *S* of Q_N such that $S \cap F_N \neq \emptyset$. That is, F_D is all sets of *N*'s states that include at least one accepting state of *N*. For each set $S \subseteq Q_N$ and for each input symbol *a* in $\sum$, $\delta_D (S,a) = \bigcup_{p\,in\,S} \delta_D(p,a)$. That is, to compute $\delta_D(S,a)$ all states *p* in *S* are checked, see what states *N* goes from *p* on input *a*, and take the union of all those states (Hopcroft et al, 2001).

1.2. Objective

This project aims to implement a console-based NFA to DFA Converter using C++ and the Dev-C++ compiler.

1.3. Scope and Limitations

The program accepts a text file as an input. The text file, which follows the format specified in Chapter 2, should contain the specifications of the NFA to be converted. The output is a text file of the same format containing the specifications of the equivalent DFA. The program can accept NFAs with any number of states and inputs. However, it is limited to accepting NFAs without ϵ-transitions. There are error notifications when there are invalid inputs.

2. Text File Format[*]

The finite automata should be defined as follows:

```
FASPEC
fa NAME;
type FATYPE;
input_alphabet ALPHABET;
states STATELIST;
transitions
{
LIST OF TRANSITION RULES;
}
fa_end;
```

FASPEC, fa, type, input_alphabet, states, transitions, and fa_end are keywords. They are case-sensitive and should come in the order specified in the format above.

NAME is an identifier serving as the name of the finite automaton. Identifiers are case-sensitive. Identifiers shall consist of sequences of one or more symbols from the identifier alphabet [a-zA-Z0-9_]. Identifiers may start with any symbol from the identifier alphabet, including the digits.

FATYPE is one of: nfa or NFA for the input file, dfa or DFA for the output file.

ALPHABET is an alphabet. The symbols allowed in alphabets are all printable characters and the space. Alphabets are case-sensitive and are specified by listing the symbols involved within square brackets. A dash may be used to indicate a range.

STATELIST is a list of identifiers separated by commas. The first identifier in the list is the start state. Identifiers with asterisks preceding them are accepting states. Identifiers are case-sensitive and shall consist of sequences of one or more symbols from the identifier alphabet [a-zA-Z0-9_]. Identifiers may start with any symbol from the

[*] This was taken from the CE 160 website. The detailed version can be seen in Appendix 1.

identifier alphabet, including the digits. The word “null”, however, can’t be used in the state list of the input file because it is a special name used for the null state, should there be one, in the DFA.

Each TRANSITION RULE is of the form: *state, symbol: statelist;*

An example is shown here. An NFA with the transition table:

States	0	1
→A	{A,B}	{A}
B	{C}	{C}
C	{D}	Ø
*D	{D}	{D}

is represented by the text file shown below.

```
FASPEC
fa myNFA;  //example
type nfa;
input_alphabet [01];
states A,B,C,*D;
transitions
{
A, 0: A, B;
A, 1: A;
B, 0: C;
B, 1: C;
C, 0: D;
D, 0: D;
D, 1: D;
}
fa_end;
```

3. Algorithms and Implementation

3.1. Program Flow

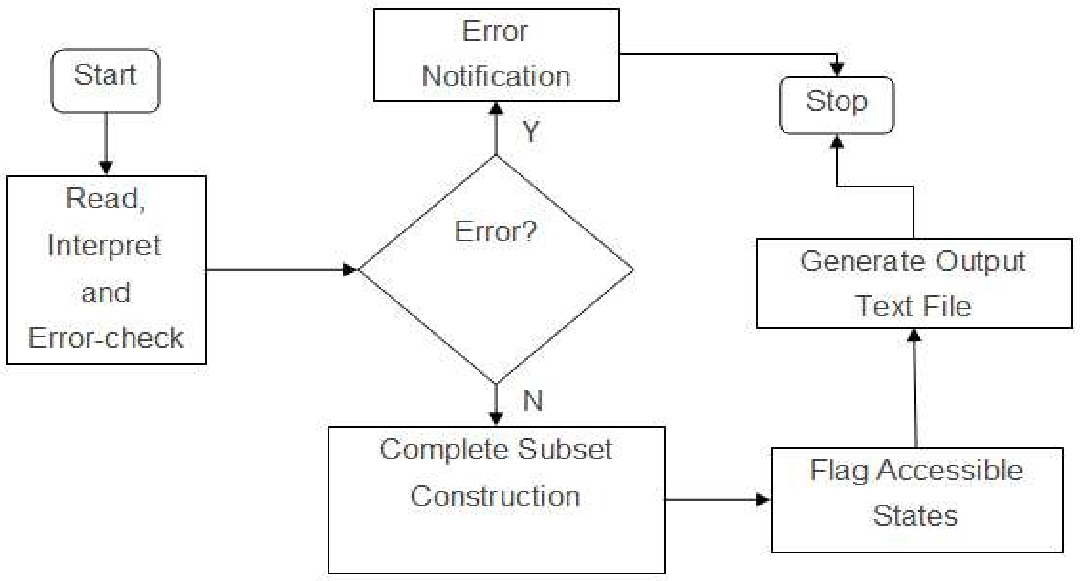

Start
Read, Interpret and Error-check
Error?
Y
N
Error Notification
Stop
Complete Subset Construction
Flag Accessible States
Generate Output Text File

Figure 3.1.1. Program Flowchart

The basic algorithm of the program is shown in the figure above. The first part is to accept the input text file. The program will check for errors, and should there be errors, the user will be prompted and the program will be terminated. If there are no errors, the program will interpret the input file and will construct all the subsets of the set of states of the NFA. States that are accessible from the start state will be marked. The last part of the program will be to output the equivalent DFA to a text file. For the state list and the transitions, only the marked states will be printed. This algorithm is carried out in three parts using eight functions, including *main*, and eight sub-functions called from the main function.

The main function creates the structure of the program and it can be separated into three basic parts: input, convert, and output. The figure below shows these parts and the major functions that are used in them.

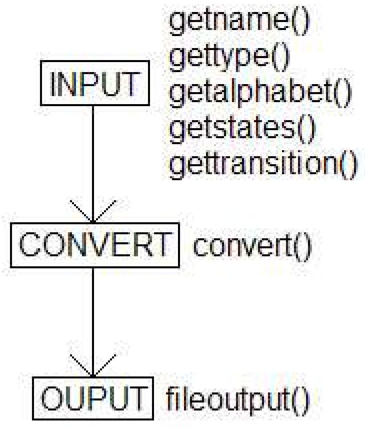


Figure 3.1.2 Three Parts of the Program with their functions

Error checking is integrated in these functions such that when an error is encountered the program will be terminated.

3.2. Input

This is the first part of the program. Here is where all the necessary data is read from the file and saved to their appropriate arrays. The first few lines simply checks the first part of the input file format, checking for keywords and getting their appropriate identifiers. Shown below is the basic code to carry out this task.

```
ictr=checkkeyword(5,"ASPEC",fread, 'F');
if(ictr==0) return 1;
ictr=getspace(fread, "FASPEC");
if(ictr==0) return 1;
```

The first line checks the file if it reads the identifier FASPEC, and the second line is for the error checking. The next two lines just ensure that there is a white space in between this keyword and the next keyword. However, if the keyword requires an identifier, another two lines are added.

```
ictr=getname(fread, &name, c);
if(ictr==0) return 1;
```

Here, the identifier for the keyword *fa* is stored in the string *name*. If there is error in acquiring this data, the program ends.

3.2.1. Input Alphabet

*int getalphabet(FILE *fread, char **alphabetlist);*

The function *getalphabet()* gets the input alphabet from the file and saves it to the character array *alphabetlist*. This function implements a dynamic memory allocation technique by first allocating an array of maximum size. This array, however, would have consumed too much memory if the alphabet was just composed of two symbols. The solution for this is to create another array of the right size and then transfer the input symbols to this new array and then dispose of the old array. The C++ operators used for this are *new* and *delete*. This function returns the number of input symbols used for the automaton.

In this function, a temporary character array of size 100 is used to store the alphabet. The process of storing the symbols is simple, but it has a lot of error checking. The first part of the code just gets the '[' character as specified by the file format. After getting this character, a for-loop is then used to get all the symbols. This loop would be terminated when the program encounters a ']' character without the proper escape sequence in the file. Of course, the program ends when an error is encountered. A part of the algorithm is shown below, there is no error checking done here.

```
temp=new char[100];
for(;input symbol!=']';) {
      if (symbol is valid) {
         assign symbol to first;
      }
      check for escape sequences and symbol duplicates;
      check if symbol='-' {
            create range of symbols and check for duplicates;
      }
      get next input symbol;
}
```

The code below checks the symbol stored in the temporary array if it is a valid symbol, meaning it should not be a white space, backslash, slash, or not the range operator. If it is a valid symbol, the program assigns that symbol to the character *first*. This is done for all valid characters read because the program is guessing that the next symbol to be read is a '-', which specifies a range. By saving the character before the dash, it would be easier for the program to create the range of symbols.

```
if(temp[alphabetnumber]!='-' && temp[alphabetnumber]!=' ' && temp[alphabetnumber]!='\n'
&& temp[alphabetnumber]!='/' && temp[alphabetnumber]!='\\') {
        first=temp[alphabetnumber];
}
```

Escape sequences are part of the file format to include symbols like the character ‘]’. These escape sequences are implemented by an if-statement and the code is shown below.

```
else if(temp[alphabetnumber]=='\\') {
        c=fgetc(fread);
        if(c=='s') temp[alphabetnumber]= ' ';
        else if(c==']') temp[alphabetnumber]= ']';
        else if(c=='[') temp[alphabetnumber]= '[';
        else if(c=='|') temp[alphabetnumber]= '|';
        else if(c==';') temp[alphabetnumber]= ';';
        else if(c==':') temp[alphabetnumber]= ':';
        else if(c=='{') temp[alphabetnumber]= '{';
        else if(c=='}') temp[alphabetnumber]= '}';
        else if(c=='\\') temp[alphabetnumber]= '\\';
        else if(c=='\"') temp[alphabetnumber]= '\"';
        else if(c=='/') temp[alphabetnumber]='/';
        first=temp[alphabetnumber];
        if(check_duplicates(temp,temp[alphabetnumber],alphabetnumber)>=0) {
           alphabetnumber--;
        }
}
```

Another function is used here to ensure that all symbols stored in the array have no duplicates. This function is *check_duplicates(char *list, char c, int max)* where **list* is the character array for which the character *c* is checked if it has any duplicates. The integer *max* just contains the height of the array. This function returns the number of the duplicate symbol in the array, if there is any, and -1 if there are no duplicates.

When all the input symbols have been saved to the temporary array and all the checks have been done, these symbols are transferred to the appropriate array, the one that was passed as an argument for the function. The temporary array is then deallocated. The code below implements this. The array *alphabetlist* would contain the alphabet and *alphabetnumber* is the number of input symbols.

```
*alphabetlist = new char[alphabetnumber];
for(ictr=0;ictr<alphabetnumber;ictr++) (*alphabetlist)[ictr]=temp[ictr];
delete temp;
```

3.2.2. NFA States

*int getstates(FILE *fread, int *statectr, int *acceptctr, int *nullctr, string **states, string **acceptstates, char c);*

Same as the *getalphabet()* function, the *getstates()* function gets the NFA states and uses dynamic memory allocation for the variables used to store it. The implemented dynamic memory allocation for the state list is the same as that used in *getalphabet()*. Allocating memory for the state list is just counting the number of states in the state list and then allocating memory for an array of strings with size equal to the number of states read before. The program first stores the states in a temporary string array. Once all the states in the input file has been read, the program allocates the right amount of memory for the string pointer **states* to point to.

**states=new string[statenum+1];*

The states are then transferred to this memory location from the temporary array. The "null" state is added and stored last.

The temporary array for the list has a maximum size of 1000. This is a safe size since NFA to DFA conversion using complete subset construction with an NFA with 1000 states can take a very big chunk of memory which may be unsafe for some computer systems with low memory. Also, the conversion process can take too long to complete.

This function is simple, but it has a lot of error checking. The basic principle here is that the program stores the non-white space characters in a string buffer until a comma or a white space is encountered. The comma basically tells the program to save the string in the string buffer to the array. Additionally, the string buffer is cleared to be able to accept the next state name. Accepting states are marked by an asterisk at the beginning of the state name. When an asterisk is read, the program remembers this by setting the character *first* to be equal to asterisk. As the program reads the characters from the file the variable first is unchanged until the program reads a semicolon or a comma. At this point the program saves the string buffer to the state array and since first is set to asterisk, the string buffer is also saved to the accepting states array. Another function is used here, *stateduplicate(string *list, string c1, int max)*. This function just checks if there is a duplicate state in the state list. This is basically the same function as *check_duplicate()* used for checking if there is a duplicate symbol in the symbols list but now used for checking duplicate states. A lot of if-statements are used here to check for all of these instances. This whole process is looped and terminated when the character ';' is read.

Shown below is a part of the algorithm used.

```
for(;c!=';';c=fgetc(fread)){
        check if character read is comma; {
        save string buffer to array;
        }
check if accepting state {
       first = asterisk;
 }
save to string buffer character read;
}
```

Once all the states have been read and saved to the temporary arrays, they are transferred to the correctly-sized arrays. The temporary arrays are then deallocated. The following code implements this last portion.

```
*states=new string[statenum+1];
*acceptstates=new string[acceptstatenum];
for(ctr=0;ctr!=statenum;ctr++) {
for(ctr=0;ctr!=acceptstatenum;ctr++) (*acceptstates)[ctr]=tempaccept[ctr];
delete tempstates;
delete tempaccept;
```

3.2.3. NFA Transitions

```
int gettransition(FILE *fread, int statenum, int acceptstatenum, int nfanullstate, int
alphabetnumber, char *alphabetlist, string *states, string *acceptstates, int ****transition, char
c);
```

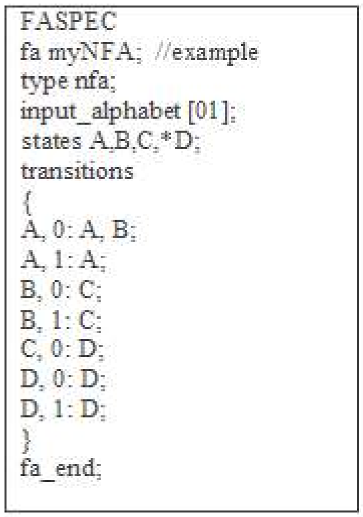
The function *gettransition()* requires a lot of arguments to be called since a lot of factors have to be considered for error checking and data storage. They are structured as such so as to make them flexible enough to be used in other programs. The file pointer **fread* points to the input file where the NFA is stored. The integer *statenum* stores the number of NFA states and *acceptstatenum* stores the number of accepting states and *nfanullstate* contains the number in the state array for the null state. The integer *alphabetnumber* contains the number of input symbols while the character pointer **alphabetlist* stores the symbols. The two string pointers **states* and *acceptstates* store the string array for states and accepting states respectively. The argument for the *int ****transition* should be the address of the pointer-to-pointer-to-pointer, or the three dimensional integer array that should contain the NFA transitions. The last argument for this function is just the last character read from *fread*.

A three dimensional integer array is used to store the NFA transitions. Numbers stored here are used to signify the position of the states and input symbol in their respective arrays. The transition array is stored as such:

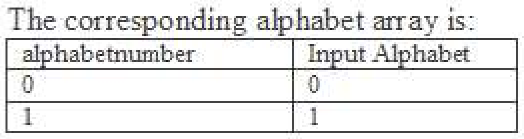
int transitions[statenumber][alphabetnumber][transitionstatenumber]

where: the statenumber signifies the position of the state in the state name array, the *alphabetnumber* signifies the position of the input symbol in the alphabet array, and the *transitionstatenumber* signifies the number of possible states it can transition to. The size of the three dimensional array is dependent on the number of states and the number of input symbols stored. Shown below is an example of how this is done.

Example 3.2.3.1. Given the NFA file below:

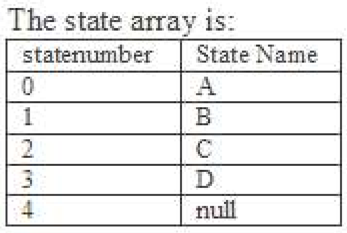


```
FASPEC
fa myNFA; //example
type nfa;
input_alphabet [01];
states A,B,C,*D;
transitions
{
A, 0: A, B;
A, 1: A;
B, 0: C;
B, 1: C;
C, 0: D;
D, 0: D;
D, 1: D;
}
fa_end;
```



The corresponding alphabet array is:

alphabetnumber	Input Alphabet
0	0
1	1



The state array is:

statenumber	State Name
0	A
1	B
2	C
3	D
4	null

The transition array is:

statenumber	alphabetnumber	transitionstatenumber	statenumber
0	0	0	0
		1	1
		2	4
		3	4
		4	4
	1	0	0
		1	4
		2	4
		3	4
		4	4
1	0	0	2
		1	4
		2	4
		3	4
		4	4
	1	0	2
		1	4
		2	4
		3	4
		4	4
2	0	0	3
		1	4
		2	4
		3	4
		4	4
	1	0	4
		1	4
		2	4
		3	4
		4	4
3	0	0	3
		1	4
		2	4
		3	4
		4	4
	1	0	3
		1	4
		2	4
		3	4
		4	4
4	0	0	4
		1	4
		2	4
		3	4
		4	4
	1	0	4
		1	4
		2	4
		3	4
		4	4

Once memory is allocated for the transition array, storing data in this array is quite simple. The program just reads the file, finds the appropriate statenumbers and alphabetnumbers for the states and input symbols, respectively, and stores the statenumbers of the states they can transition to. Almost the same algorithm is used here as that used in *getstates()* and *getalphabet()* for getting the state and for getting the input symbol, respectively. If there is no transition for a given state and symbol, the *statenumber* for the null state is stored. Shown below is the basic algorithm used.

```
for(;'}';) {
    get state; check if state exist;
    get symbol; check if symbol exists;
    for(;';';) {
        get transition states;check if state exists;
        save to transitions[][][];
    }
}
```

In this function, the two functions to check for duplicates in symbols and states are used to check if the symbol or state exists in the given alphabet and state list respectively. Take for example, the code below. This code just checks if the string in buffer has been declared as a valid state in the state list. If the string in *buffer* is not a valid state the program would present an error statement and end.

```
ictr=stateduplicate(states, buffer, statenum);
if(ictr<0) {
            cout<<"\nCannot Find State! - transitions. Press any key to quit.";
           getch();
           return 0;
}
```

3.3. Complete Subset Construction

```
int convert(string **complete, string **completeaccept, string ***completetransition, string *states, string *acceptstates, int **passflag, int *passtemp, int *completetemp, int statenum, int alphabetnumber, int nfanullstate, int acceptstatenum, int ***transition, char *alphabetlist);
```

The program uses the complete subset construction to convert the NFA to its equivalent DFA. This is another big function requiring a lot of arguments and the reason for this is the same, flexibility for this function to be reused in other functions. Aside from those previously used variables, this function introduces six more variables. The string pointers arguments ***complete* and ***completeaccept* should contain the addresses for the one-dimensional string array that should be receiving the new state names and accepting state names derived from the complete subset construction. The third argument should contain the address of the two-dimensional string array that should receive the complete subset transitions. The integer pointer-to-pointer argument ***passflag* should contain the address for a one-dimensional integer array that will contain the flags used to determine whether the subset state is a reachable state. The next two arguments should contain the addresses for the integers that would be containing the number of reachable states and the total number of states generated respectively.

3.3.1. State Names

The idea is to use a binary code to generate all the possible combinations of the NFA states since the states in the complete subset construction are just combinations of the NFA states. How is this done? First, the maximum number of DFA states must be computed. This is just equal to 2^n, where n is the number of NFA states. The next step is to convert 0 to 2^n-1 into reverse binary (using modulo 2) and store it in an array. The following code is used to do this:

```
for(i=0;i<completenum;i++) {
      l=i;
      j=i;
      for(k=0;k<statenum;k++) {
         l=l/2;
         j=j%2;
         if (j==0) bincode[ctr]="0";
         else bincode[ctr]="1";
         ctr++;
         j=l;
      }
      bincode[ctr]=" ";
      ctr++;
}
```

For example, if the number of NFA states is 3, there is a maximum of 8 (2^3) DFA states and the reverse binary code for 0-7 are shown below.

Table 3.3.1.1. Reverse Binary Code for NFA States = 3

Number	Reverse Binary Code (no. of digits = no. of NFA states)
0	000
1	100
2	010
3	110
4	001
5	101
6	011
7	111

These 8 reverse binary combinations represent the 8 combinations of the NFA states. The first digit represents the first NFA state, which is the start state. The second and third digits represent the second and third NFA states saved in the array of states, respectively. These codes are saved in the temporary array in this manner:

000 100 010 110 001 101 011 111

The program will then read the array. When it sees a 0, it means that the corresponding state of the digit is not included in the new state name. On the other hand, when it sees a 1, it appends the corresponding state of the digit to the string buffer used to combine the state names. After reading a line of reverse binary code and the string buffer does not contain anything (for the code 000), the state name will be “null”. If, for example, there are three NFA states – Q1, Q2 and Q3 – the new state names for the complete subset construction are shown in the Table 3.3.1.2.

Table 3.3.1.2. Reverse Binary Code for NFA States = 3 and Corresponding State Name

Number	Reverse Binary Code (no. of digits = no. of NFA states)	State Name
0	000	null
1	100	Q1
2	010	Q2
3	110	Q1_Q2
4	001	Q3
5	101	Q1_Q3
6	011	Q2_Q3
7	111	Q1_Q2_Q3

After reading a line of binary code, the string buffer is saved in a new array of states (array for the complete subset construction). The string buffer is then refreshed and the array is incremented. The program stops reading when it sees a semicolon. By then, all of the combinations of the states names of the NFA have already been saved in the new array of states.

3.3.2. Transitions

The "computation" for the transitions of the states (after the complete subset construction) is done in parallel with the generation of the state names. Remember that the new state names are created by a reverse binary code. Whenever the program reads a 1, it appends the corresponding state of the binary digit being read to a string buffer. After this, the transitions of that state are saved in a two-dimensional temporary array:

int temporary[alphabetnumber][completenum]

where: *alphabetnumber* signifies the position of the input symbol in the symbol array and *completenum* is the maximum number of states. The idea is to copy to an array all the states that a particular state can transition to, at every symbol. This is done because the transition of the new state (generated from the binary code) is just the union of all the individual states. After copying all the transitions of all the individual states of the new state in an array, these are combined in a string buffer, each state separated by an underscore. The following pseudocode is a simplified implementation of this algorithm.

```
for(all characters in bincode array) {
    if bincode is " "{
        add state name to complete array;
        create transition state from states in temporary array;
    }
    else if  bincode is 1 {
        add NFA state name to buffer;
        add NFA transitions of the state for all symbols to temporary array;
    }
}
```

3.3.3. Marking the Accessible States

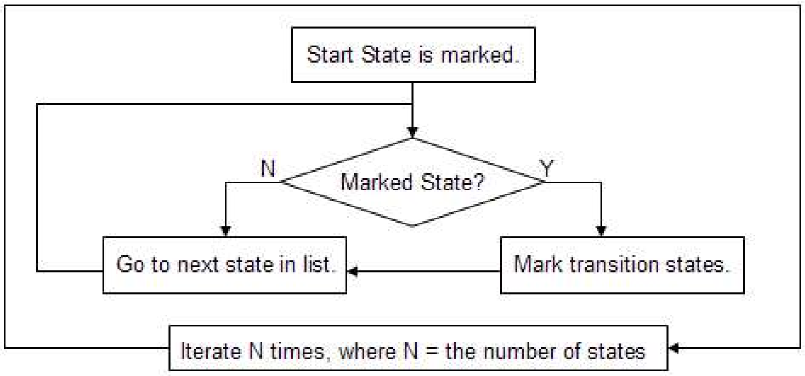

Start State is marked.
Marked State?
N
Y
Go to next state in list.
Mark transition states.
Iterate N times, where N = the number of states

Figure 3.3.3.1. Marking the Accessible States

Marking the accessible states from the start state is quite simple. The idea is to begin from the start state and mark the states in which it can transition to. Next is to move on through the state list and check state per state if they are marked or not. If the state is marked, mark the states it can transition to. If not, move on to the next state. This process should be iterated N times where N is the number of states. This is to ensure that all accessible states are marked. The following code is used to implement this.

```
for(i=0;i<completenum;i++) {
    for(j=0;j<completenum;j++) {
        for(k=0;k<alphabetnumber;k++) {
            if((*passflag)[j]==1) {
                l=stateduplicate(*complete,(*completetransition)[j][k],completenum);
                if((*passflag)[l]==1);
                else passnum++;
                (*passflag)[l]=1;
            }
        }
    }
}
```

The example below shows how this algorithm works.

Example 3.3.3.1. The transition table below shows the complete subset construction of an NFA.

States	0	1
→p	q_r_p	q_r_s
*q	r	q_r
r	s	p
*s	null	q
*q_s	r	q_r_p
*q_r	r_s	q_r_p
*q_p	q_r_s	q_r
r_p	q_s	q_p
*r_s	s	p
*p_s	q_s	q_p
*p_r_s	q_s	q_p
*q_r_p	q_r_s	q_r_p
*p_q_s	q_p_r_s	q_r_s
*q_r_s	r_s	q_r_p
*q_p_r_s	q_r_s	q_r_p
null	null	null

The first iteration would mark the states in the following order: p, q_r_p, q_r_s and r_s. The process, however, is continued since the transition states for r_s are not yet marked. The second iteration then marks s and p. The third iteration marks null and q. The fourth iteration marks r and q_r. The next iterations won't mark any more states since all accessible states are already marked. The resulting DFA transition table using the marked states is shown below.

States	0	1
→p	q_r_p	q_r_s
*q	r	q_r
r	s	p
*s	null	q
*q_r	r_s	q_r_p
*r_s	s	p
*q_r_p	q_r_s	q_r_p
*q_r_s	r_s	q_r_p
null	null	null

3.4 File Output

*int fileoutput(char *fname, string name, string *complete, string *completeaccept, string **completetransition, int *passflag, int passnum, int alphabetnumber, int completenum, char *alphabetlist);*

This last part prints out to the output file the constructed DFA. This is the simplest part of the program consisting only of printing to file and virtually no need for error checking save for the availability of the output file. Here, almost all of the required arguments are just those derived form the complete subset construction. The only new argument here is the character pointer **fname* containing the name of the text file that the function should be writing to. Also, the NFA name is retained for the DFA name that appears in the output text file. Escape sequences are implemented and all symbols are printed back to their respective escape sequences.

4. Results

The first four NFAs that will be shown here were derived from Hopcroft's Introduction to Automata Theory, Languages and Computation. All of the input and output files of the tests shown here are also included in the CD.

4.1. Example 2.7

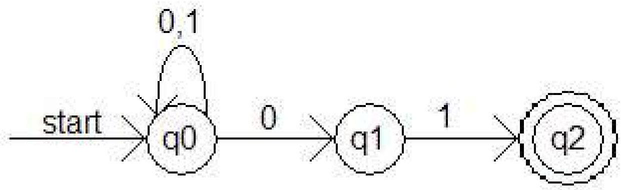


Figure 4.1.1. NFA specified in Example 2.7

An equivalent DFA using complete subset construction is shown below.

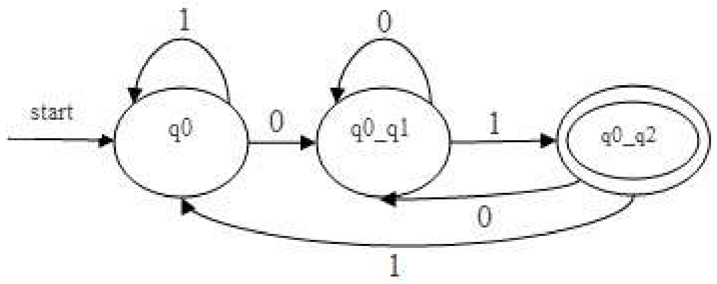


Figure 4.1.2. A DFA equivalent of the Figure 4.1.1

The NFA in Figure 4.1.1. is represented in the input file below.

```
FASPEC
fa example_2_7;
type NFA;
input_alphabet [01];
states  q0,q1,*q2 ;
transitions
{
q0,0:q0,q1;
q0,1:q0;
q1,1:q2;
}
fa_end;
```

The resulting output file from the program for the given input file is shown below.

```
FASPEC
fa example_2_7;
type DFA;
input_alphabet [ 0  1 ];
states        q0,        q0_q1,
*q0_q2 ;
transitions
{
q0 , 0 : q0_q1 ;
q0 , 1 : q0 ;
q0_q1 , 0 : q0_q1 ;
q0_q1 , 1 : q0_q2 ;
q0_q2 , 0 : q0_q1 ;
q0_q2 , 1 : q0 ;
}
fa_end;
```

The DFA generated by the program is the same DFA shown in Figure 4.1.2. This shows that the program has generated a correct DFA for the given NFA. The program is able to create this output file within a second.

4.2. Exercise 2.3.1

The NFA specified in the exercise is shown below.

Exercise 2.3.1 NFA Transition Table

	0	1
→p	{p,q}	{p}
q	{r}	{r}
r	{s}	Ø
*s	{s}	{s}

An equivalent DFA has the transitions:

	0	1
→p	p_q	p
p_q	p_q_r	p_r
p_r	p_q_s	p
*p_s	p_q_s	p_s
p_q_r	p_q_r_s	p_r
*p_q_s	p_q_r_s	p_r_s
*p_r_s	p_q_s	p_s
*p_q_r_s	p_q_r_s	p_r_s

The NFA of Exercise 2.3.1 is represented in the input file below.

```
FASPEC
fa exer_2_3_1;
type NFA;
input_alphabet [01];
states  p,q,r,*s ;
transitions
{
p,0:p,q;
p,1:p;
q,0:r;
q,1:r;
r,0:s;
s,0:s;
s,1:s;
}
fa_end;
```

The DFA generated by the program is shown below. The output is correct since it represents the DFA shown before.

```
FASPEC
fa exer_2_3_1;
type DFA;
input_alphabet [ 0  1 ];
states  p,  p_q,  p_r,  p_q_r,  *p_s,  *p_q_s,  *p_r_s,  *p_q_r_s ;
transitions
{
p , 0 : p_q ;
p , 1 : p ;
p_q , 0 : p_q_r ;
p_q , 1 : p_r ;
p_r , 0 : p_q_s ;
p_r , 1 : p ;
p_q_r , 0 : p_q_r_s ;
p_q_r , 1 : p_r ;
p_s , 0 : p_q_s ;
p_s , 1 : p_s ;
p_q_s , 0 : p_q_r_s ;
p_q_s , 1 : p_r_s ;
p_r_s , 0 : p_q_s ;
p_r_s , 1 : p_s ;
p_q_r_s , 0 : p_q_r_s ;
p_q_r_s , 1 : p_r_s ;
}
fa_end;
```

It took, more or less, one second for the program to generate this output file.

4.3. Exercise 2.3.2

The NFA specified in the in the exercise is shown below.

	0	1
→p	{q,s}	{q}
*q	{r}	{q,r}
r	{s}	{p}
*s	Ø	{p}

When converted to DFA the transition table is shown below.

	0	1
Ø	Ø	Ø
→ p	q_s	q
*q	r	q_r
r	s	p
*s	Ø	p
*q_s	r	p_q_r
*q_r	s_r	p_q_r
*s_r	s	p
*p_q_r	q_r_s	p_q_r
*q_r_s	s_r	p_q_r

The NFA above is represented in the input file below.

```
FASPEC
fa exer2_3_2;
type NFA;
input_alphabet [01];
states  p,*q,r,*s ;
transitions
{
p,0:q,s;
p,1:q;
q,0:r;
q,1:q,r;
r,0:s;
r,1:p;
s,1:p;
}
fa_end;
```

The DFA generated by the program is shown below. The output is correct since it is represents the DFA shown before.

```
FASPEC
fa exer2_3_2;
type DFA;
input_alphabet [ 0  1 ];
states  p,  *q,  r,  *q_r,  *p_q_r,  *s,  *q_s,  *r_s,  *q_r_s,  null ;
transitions
{
null , 0 : null ;
null , 1 : null ;
p , 0 : q_s ;
p , 1 : q ;
q , 0 : r ;
q , 1 : q_r ;
r , 0 : s ;
r , 1 : p ;
q_r , 0 : r_s ;
q_r , 1 : p_q_r ;
p_q_r , 0 : q_r_s ;
p_q_r , 1 : p_q_r ;
s , 0 : null ;
s , 1 : p ;
q_s , 0 : r ;
q_s , 1 : p_q_r ;
r_s , 0 : s ;
r_s , 1 : p ;
q_r_s , 0 : r_s ;
q_r_s , 1 : p_q_r ;
}
fa_end;
```

It took one second, more or less, for the program to generate this output file.

4.4. Exercise 2.3.3

The NFA specified in the exercise is shown below.

	0	1
→p	{p,q}	{p}
*q	{r,s}	{t}
r	{p,r}	{t}
*s	Ø	Ø
*t	Ø	Ø

When converted to a DFA, the transition table is shown below.

	0	1
→p	p_q	p
p_q	p_q_r_s	p_t
*p_t	p_q	p
*p_q_r_s	p_q_r_s	p_t

The NFA above is represented in the input file below.

```
FASPEC
fa exer2_3_3;
type NFA;
input_alphabet [01];
states  p,q,r,*s,*t ;
transitions
{
p,0:p,q;
p,1:p;
q,0:r,s;
q,1:t;
r,0:p,r;
r,1:t;
}
fa_end;
```

The DFA generated by the program is shown below. The output is correct since it is represents the DFA shown before.

```
FASPEC
fa exer2_3_3;
type DFA;
input_alphabet [ 0  1 ];
states  p,  p_q,  *p_q_r_s,  *p_t ;
transitions
{
p , 0 : p_q ;
p , 1 : p ;
p_q , 0 : p_q_r_s ;
p_q , 1 : p_t ;
p_q_r_s , 0 : p_q_r_s ;
p_q_r_s , 1 : p_t ;
p_t , 0 : p_q ;
p_t , 1 : p ;
}
fa_end;
```

It took one second, more or less, for the program to generate this output file.

4.5 Other Tests

Another example from the book was used to test the program's capabilities. This is Example 2.14. This NFA recognize occurences of the words "ebay" and "web". This NFA has eight states and 26 input symbols. An equivalent DFA, which is also shown in the book, has also 8 states, two of which are accepting. The program output is the DFA shown in the book.

An NFA with the language that accepts all strings of 0's and 1's that has the symbol 1 as its ninth symbol from the right, was used to test the program. The problem with this test, however, is the fact that there is no way to verify a very big DFA output. The only logical proof that the output DFA is correct is the fact that the program works for smaller NFA to DFA conversions. The representation of the NFA used in the input file is shown in the following page.

```
FASPEC
fa TEST;
type NFA;
input_alphabet [ 01 ]; //comment
states  1, 2, 3, 4, 5, 6, 7, 8, 9, *10;
transitions
{
1,0:1;
1,1:1,2;
2,0:3;
2,1:3;
3,0:4;
3,1:4;
4,0:5;
4,1:5;
5,0:6;
5,1:6;
6,0:7;
6,1:7;
7,0:8;
7,1:8;
8,0:9;
8,1:9;
9,0:10;
9,1:10;
}
fa_end;
```

The program generated a 40kB text file in approximately 2 minutes of run time. The output text file is saved in the CD as "sir2.txt".

Consequently, the same NFA was extended to "tenth symbol from the right". This NFA generated an 86kB text file in approximately 11 minutes of run time. This output text file is saved in the CD as "sirdfa.txt".

5. Recommendations

Improvements can be done to enhance the efficiency of this program. One enhancement would be to interpret any unexpected text in the input file as a comment. An improvement like this should consider the fact that a lot of data are needed for a successful conversion. These data should not be compromised whenever an unexpected text is dismissed as a comment. Another improvement would be to take advantage of C++'s capabilities for object oriented programming instead of a procedural one. Object oriented programming make C++ codes more structured, thus, making it easier to understand epecially to those familiar with the language. Another improvement would be to use the lazy evalutation instead of complete subset construction to derive an equivalent DFA of the input NFA. This reduces the memory required for conversion.

One problem with this program, however, is that it cannot verify the resulting DFA if it has a lot of states. A solution would be to create a DFA simulator program to verify the DFA output of this program.

Appendix 1: Text File Format[*]

The finite automata should be defined as follows:

```
FASPEC
fa NAME;
type FATYPE;
input_alphabet ALPHABET;
states STATELIST;
transitions
{
LIST OF TRANSITION RULES;
}
fa_end;
```

FASPEC, fa, type, input_alphabet, states, transitions, and fa_end are keywords. They are case-sensitive and should come in the order specified in the format above.

NAME is an identifier serving as the name of the finite automaton. Identifiers are case-sensitive. Identifiers shall consist of sequences of one or more symbols from the identifier alphabet [a-zA-Z0-9_]. Identifiers may start with any symbol from the identifier alphabet, including the digits.

FATYPE is one of: nfa or NFA for the input file, dfa or DFA for the output file.

ALPHABET is an alphabet. The symbols allowed in alphabets are those on which isgraph() returns true, and the space. Alphabets are case-sensitive and are specified by listing the symbols involved within square brackets:
e.g.

- [01] for the binary alphabet
- [0 1 2 3 4 a b c] for the set {0,1,2,3,4,a,b,c}
- [( )] for the set consisting of parentheses
- [()\s] is a set with three symbols (a space and left and right parentheses)

[*] This was taken from the CE 160 website.

A dash may be used to indicate a range:

e.g.

- [0-9] for the set of 10 decimal digits
- [A-Wa] for the set of uppercase letters from A to W plus lowercase a
- [0-9a-f] for the set of hexadecimal digits, but not including uppercase A to F

If a dash is part of the alphabet, it must be specified either first or last:

e.g.

- [-+*/] for the set {-,+,*,/}

The following escape sequences are recognized as part of the alphabet:

- \s for the space symbol
- \] for the right square bracket ‘]’
- \[for the left square bracket ‘[‘
- \| for the pipe ‘|’
- \/ for the slash ‘/’
- \; for the semicolon ‘;’
- \: for the colon ‘:’
- \{ for the left curly brace ‘{‘
- \} for the right curly brace ‘}’
- \\ for the backslash
- \“ for double quotes

STATELIST is a list of identifiers separated by commas. The first identifier in the list is the start state. Identifiers with asterisks preceding them are accepting states. Identifiers are case-sensitive and shall consist of sequences of one or more symbols from the identifier alphabet [a-zA-Z0-9_]. Identifiers may start with any symbol from the identifier alphabet, including the digits. The word “null”, however, can’t be used in the state list of the input file because it is a special name used for the null state, should there be one, in the DFA.

Each TRANSITION RULE is of the form:

state, symbol: statelist;

In transition rules (symbol), the following are used for the input and output files:

- \s for the space symbol (a must in the input file)
- \] for the right square bracket ‘]’
- \[for the left square bracket ‘[‘
- \| for the pipe ‘|’
- \/ for the slash ‘/’
- \; for the semicolon ‘;’
- \: for the colon ‘:’ (a must in the input file)
- \{ for the left curly brace ‘{‘
- \} for the right curly brace ‘}’
- \\ for the backslash (a must in the input file)
- \“ for double quotes

Comments starting with // (double slash) and white spaces in the input file are ignored.

Appendix 2: Code

```
#include <conio.h>
#include <iostream>
#include <string>
using namespace std;

//Functions
int comment(FILE *fread);
int check(char c);
int check_duplicates(char *list, char c, int max);
int stateduplicate(string *list, string c1, int max);
int sorttransition(int list[], int max);
int checkkeyword(int keywordlen, string keyword, FILE *fread, char c1);
char lookforchar(FILE *fread, string str);
int getname(FILE *fread, string *namae, char c);
int gettype(FILE *fread);
int getspace(FILE *fread, string where);
int getalphabet(FILE *fread, char **alphabetlist);
int getstates(FILE *fread, int *statectr, int *acceptctr, int *nullctr, string **states, string **acceptstates, char
c);
int inittransit(int ****transition, int statenum, int alphabetnumber);
int gettransition(FILE *fread, int statenum, int acceptstatenum, int nfanullstate, int alphabetnumber, char
*alphabetlist, string *states, string *acceptstates, int ****transition, char c);
int fileoutput(char *fname, string name, string *complete, string *completeaccept, string
**completetransition, int *passflag, int passnum, int alphabetnumber, int completenum, char *alphabetlist);
int convert(string **complete, string **completeaccept, string ***completetransition, string *states, string
*acceptstates, int **passflag, int *passtemp, int *completetemp, int statenum, int alphabetnumber, int
nfanullstate, int acceptstatenum, int ***transition, char *alphabetlist);

//Main Function
int main(int argc, char *argv[])
{
   char c, *alphabetlist;
   string name,*complete, *completeaccept, **completetransition;;
        int ***transition, statenum, acceptstatenum, ictr, alphabetnumber, **dfatransition, nfanullstate,
completenum, *passflag, passnum ;

   //file input sequence start
   if(argc!=3) {
      cout<<"\nError! You have to enter two filenames. Press <enter> to quit.";
      getch();
      return 1;
   }
   FILE *fread = fopen(argv[1],"r");
   if(fread==NULL) {
      cout<<"\nError! Cannot open file! Press any key to quit.";
      getch();
      return 1;
   }
   ictr=checkkeyword(5,"ASPEC",fread, 'F');  //The following lines check the file format of the input file
   if(ictr==0) return 1;                     //and getting the necessary data such as automata name and type
   ictr=getspace(fread, "FASPEC");
```

```
    if(ictr==0) return 1;
    ictr=checkkeyword(1,"a",fread, 'f');
    if(ictr==0) return 1;
    ictr=getspace(fread, "fa");
    if(ictr==0) return 1;
    c=lookforchar(fread, "fa");
    if(c==0) return 1;
    ictr=getname(fread, &name, c);
    if(ictr==0) return 1;
    ictr=checkkeyword(3,"ype",fread,'t');                    //This program only accepts NFA types
    if(ictr==0) return 1;
    ictr=gettype(fread);
    if(ictr==0) return 1;
    ictr=checkkeyword(13,"nput_alphabet",fread,'i');
    if(ictr==0) return 1;
    ictr=getspace(fread, "input_alphabet");
    alphabetnumber=getalphabet(fread, &alphabetlist);        //Get alphabet.
    if(alphabetnumber==0) return 1;
    ictr=checkkeyword(5,"tates",fread,'s');
    if(ictr==0) return 1;
    ictr=getspace(fread, "states");
    if(ictr==0) return 1;
    c=lookforchar(fread, "states");
    if(c==0) return 0;
    string *states, *acceptstates;
    ictr=getstates(fread, &statenum, &acceptstatenum, &nfanullstate, &states, &acceptstates, c);
    //Get states.
    if(ictr==0) return 1;
    ictr=checkkeyword(10,"ransitions",fread,'t');
    if(ictr==0) return 1;
    ictr=getspace(fread, "transitions");
    if(ictr==0) return 1;
    c=lookforchar(fread, "transitions");
    if(c==0) return 1;
    ictr=inittransit(&transition, statenum, alphabetnumber);
    if(ictr==0) return 1;                           //Get transitions.
    ictr=gettransition(fread,statenum,acceptstatenum,nfanullstate,alphabetnumber,alphabetlist,states,
    acceptstates,&transition,c);
    if(ictr==0) return 1;
    ictr=checkkeyword(5,"a_end",fread,'f');
    if(ictr==0) return 1;
    ictr=checkkeyword(0,"",fread, ';');
    if(ictr==0) return 1;
    fclose(fread);
    //file input sequence end

    //dfa conversion start
        ictr=convert(&complete, &completeaccept, &completetransition, states, acceptstates, &passflag,
&passnum, &completenum, statenum, alphabetnumber, nfanullstate, acceptstatenum, transition,
alphabetlist);
    if(ictr==0) return 1;
    //dfa conversion end


    //file output sequence start
        ictr=fileoutput(argv[2], name, complete, completeaccept, completetransition, passflag, passnum,
alphabetnumber, completenum, alphabetlist);
```

```
    if(ictr==0) return 1;
    //file output sequence end

    cout<<"NFA to DFA conversion complete! Press any key to quit.";
    getch();
    return 1;
}

//Function: comment()
//this brings the file cursor to the next line
int comment(FILE *fread)
{
    char c;
    for(c=fgetc(fread);c!='\n' && c!=EOF;c=fgetc(fread));
    return 1;
}

//Function: check_duplicates()
//checks the character list list[max] if there is already a character c saved in it. Returns the position if
//there is and -1 if there is none.
int check_duplicates(char *list, char c, int max)
{
    int ictr;
    for(ictr=0;ictr!=max;ictr++) {
        if(c==*(list+ictr)) return ictr;
    }
    return -1;
}

//Function: check()
//checks if the character c is an uppercase letter, a lowercase letter, a number, or an underscore.
//Returns 0 if not any.
int check(char c)
{
    if(c>='A' && c<='Z') return 1;
    else if(c>='a' && c<='z') return 2;
    else if(c>='0' && c<='9') return 3;
    else if(c=='_') return 4;
    else return 0;
}

//Function: stateduplicate()
//checks if there is a string c1 saved in the string array list[max]. Returns position if there is and -1 if
//there is none.
int stateduplicate(string *list, string c1, int max)
{
    int ctr;
    for(ctr=0;ctr!=max;ctr++) {
        if(list[ctr]==c1) return ctr;
    }
    return -1;
}

//Function: sorttransition()
//rearranges the contents of list[max] to ascending order (in terms of state number)
int sorttransition(int list[], int max) {
    int temp, ictr, ctr;
    for(ictr=0;ictr<max-1;ictr++) {
        for(ctr=0;ctr<max-1;ctr++) {
            if(*(list+ctr)>*(list+ctr+1)) {
                temp=*(list+ctr);
                *(list+ctr)=*(list+ctr+1);
                *(list+ctr+1)=temp;
            }
            else if(*(list+ctr)==*(list+ctr+1)) *(list+ctr+1)=max;
        }
    }
}
```

//Function: checkkeyword()
//looks for the keyword given in the argument with length keywordlen in fread. c1 is the first letter of // the keyword.

int checkkeyword(int keywordlen, string keyword, FILE *fread, char c1)

```
{
    char c;
    int i;
    string keywordbuffer;
    keywordbuffer="";
    for(c=fgetc(fread);c!=c1;c=fgetc(fread)) {
        if(c==EOF){
            cout<<"\nIncomplete File Error! Press any key to quit.";
            getch();
            return 0;
        }
        else if(c=='/') {
            c=fgetc(fread);
            if(c!='/') {
                cout<<"\nKeyword Error! - "<<c1<<keyword<<". Press any key to quit.";
                getch();
                return 0;
            }
            comment(fread);
        }
        else if(c=='\n' || c==' ');
        else {
            cout<<"\nKeyword Error! - "<<c1<<keyword<<". Press any key to quit.";
            getch();
            return 0;
        }
    }
    for(i=0;i<keywordlen;i++) {
        c=fgetc(fread);
        keywordbuffer+=c;
    }
    if (keywordbuffer!=keyword) {
        cout<<"\nKeyword Error! - "<<c1<<keyword<<". Press any key to quit.";
        getch();
        return 0;
    }
    return 1;
}
```

//Function: lookforchar()
//disregards white spaces as it looks for the first non-white space character in fread. returns that // character.

char lookforchar(FILE *fread, string str)

```
{
   char c;
   for(c=fgetc(fread);c==' ' || c=='\n' || c=='/';c=fgetc(fread)) {
      if(c==EOF) {
         cout<<"\nIncomplete File Error! Press any key to quit.";
         getch();
         return 0;
      }
      else if(c=='/') {
         c=fgetc(fread);
         if(c!='/') {
            cout<<"\nSyntax error after "<<str<<"! Press any key to quit.";
            getch();
            return 0;
         }
         comment(fread);
      }
   }
   return c;
}
```

//Function: getname()
//Gets the automaton name from fread

int getname(FILE *fread, string *namae, char c)

```
{
   string name;
   for(;c!=';';c=fgetc(fread)){
      if(c==EOF){
         cout<<"\nIncomplete File Error! Press any key to quit.";
         getch();
         return 0;
      }
      else if(c==' ' || c=='\n' || c=='/') {
         for(c=fgetc(fread);c!=';';c=fgetc(fread)) {
            if(c!=' ' && c!='\n' && c!='/') {
               cout<<"\nIdentifier Error! - ;. Press any key to quit.";
               getch();
               return 0;
            }
            else if(c=='/') {
               c=fgetc(fread);
               if(c!='/') {
                  cout<<"\nIdentifier Error! - ;. Press any key to quit.";
                  getch();
                  return 0;
               }
               comment(fread);
            }
         }
         break;
      }
      else if(check(c)==0)  {
            cout<<"\nIdentifier error! Only letters, numbers and underscore are accepted. Press any key to
quit.";
         getch();
         return 0;
      }
      else {
         name+=c;
      }
   }
   *namae=name;
   return 1;
}
```

//Function: gettype()
//gets automaton type from fread. only accepts NFA type or else gives an error and returns 0.
int gettype(FILE *fread)

```
{
   char c;
   int ictr;
   c=fgetc(fread);
   if(c!=' ' && c!='\n') {
      cout<<"\nSyntax error after type! Press any key to quit";
      getch();
      return 0;
   }
   for(c=fgetc(fread);c!='N' && c!='n';c=fgetc(fread)) {
      if(c==EOF){
         cout<<"\nIncomplete File Error! Press any key to quit";
         getch();
         return 0;
      }
      else if(c=='/') {
         c=fgetc(fread);
         if(c!='/') {
            cout<<"\nSyntax error after type! Press any key to quit.";
            getch();
            return 0;
         }
         comment(fread);
      }
      else if(c==' ' || c=='\n');
      else {
         cout<<"\nIdentifier Error! - NFA type only. Press any key to quit";
         getch();
         return 0;
      }
   }
   c=fgetc(fread);
   if(c!='F' && c!='f') {
      cout<<"\nIdentifier Error! - NFA type only. Press any key to quit";
      getch();
      return 0;
   }
   c=fgetc(fread);
   if(c!='A' && c!='a') {
      cout<<"\nIdentifier Error! - NFA type only. Press any key to quit";
      getch();
```

```
        return 0;
    }
    ictr=checkkeyword(0,"",fread, ';');
    if(ictr==0) return 0;
    return 1;
}

//Function: getspace()
//just gets a white space. used for separation of keywords and/or identifiers.
int getspace(FILE *fread, string where)
{
    char c;
    c=fgetc(fread);
    if(c!=' ' && c!='\n') {
        cout<<"\nSyntax error after "<<where<<"! Press any key to quit.";
        getch();
        return 0;
    }
    return 1;
}

//Function: getalphabet()
//gets the alphabet from fread. stores them in alphabetlist[alphabetnumber].
int getalphabet(FILE *fread, char **alphabetlist)
{
    int ictr, alphabetnumber;
    char *temp, first, a, c;
    if(ictr==0) return 0;
    ictr=checkkeyword(0,"",fread, '[');
    if(ictr==0) return 0;
    alphabetnumber=0;
    temp=new char[100];
    if(temp==NULL) {
        cout<<"\nOut of Memory Error! Press any key to quit.";
        getch();
        return 0;
    }
    temp[alphabetnumber]=fgetc(fread);
    for(;temp[alphabetnumber]!=']';){
        if(temp[alphabetnumber]==EOF){
            cout<<"\nIncomplete File Error! Press any key to quit.";
            getch();
            return 0;
        }
        if(temp[alphabetnumber] == '\n');
        else if(temp[alphabetnumber]<0x20 || temp[alphabetnumber]>0x7e) {
                cout<<"\nInput Alphabet Error! - Non-printable character. Press any key to quit.";
                getch();
                return 0;
        }
            if(temp[alphabetnumber]!='-' && temp[alphabetnumber]!=' ' && temp[alphabetnumber]!='\n' &&
temp[alphabetnumber]!='/' && temp[alphabetnumber]!='\\') {
            first=temp[alphabetnumber];
        }
        if(temp[alphabetnumber]=='\n' || temp[alphabetnumber]==' ') alphabetnumber--;
        else if(temp[alphabetnumber]=='/') {
```

```
    c=fgetc(fread);
    if(c=='/'){
       comment(fread);
       alphabetnumber--;
    }
    else {
       cout<<"\nInput Alphabet Error! Press any key to quit.";
       getch();
       return 0;
    }
}
else if(temp[alphabetnumber]=='\\') {            //check for escape sequences
    c=fgetc(fread);
    if(c=='s') temp[alphabetnumber]= ' ';
    else if(c==']') temp[alphabetnumber]= ']';
    else if(c=='[') temp[alphabetnumber]= '[';
    else if(c=='|') temp[alphabetnumber]= '|';
    else if(c==';') temp[alphabetnumber]= ';';
    else if(c==':') temp[alphabetnumber]= ':';
    else if(c=='{') temp[alphabetnumber]= '{';
    else if(c=='}') temp[alphabetnumber]= '}';
    else if(c=='\\') temp[alphabetnumber]= '\\';
    else if(c=='\"') temp[alphabetnumber]= '\"';
    else if(c=='/') temp[alphabetnumber]='/';
    first=temp[alphabetnumber];
    if(check_duplicates(temp,temp[alphabetnumber],alphabetnumber)>=0) {
       alphabetnumber--;
    }
}
else if(temp[alphabetnumber]=='-') {            //check for range operation
    if(alphabetnumber==0);
    else {
       c=fgetc(fread);
       for(;c==' ' || c=='\n' || c=='/';c=fgetc(fread)) {
          if(c=='/') {
             a=fgetc(fread);
             if(a=='/') comment(fread);
             else {
                cout<<"\nInput Alphabet Error! - Range. Press any key to quit.";
                getch();
                return 0;
             }

          }
       }
       a=first;
       if(c==']') {
          if(check_duplicates(temp,temp[alphabetnumber],alphabetnumber)>=0) {
             alphabetnumber--;
          }
          alphabetnumber++;
          break;
       }
       else if(c=='\\') {
          c=fgetc(fread);
          if(c=='s') c= ' ';
          else if(c==']') c= ']';
```

```
                    else if(c=='[') c= '[';
                    else if(c=='|') c= '|';
                    else if(c==';') c= ';';
                    else if(c==':') c= ':';
                    else if(c=='{') c= '{';
                    else if(c=='}') c= '}';
                    else if(c=='\\') c= '\\';
                    else if(c=='\"') c= '\"';
                    else if(c=='/') c='/';
                    else {
                        cout<<"\nInput Alphabet Error! - Range. Press any key to quit.";
                        getch();
                        return 0;
                    }
                }
                if(a<c){
                    alphabetnumber--;
                    for(a++;a!=c+1;a++) {
                        alphabetnumber++;
                        temp[alphabetnumber]=a;
                        if(check_duplicates(temp,temp[alphabetnumber],alphabetnumber)>=0) {
                            alphabetnumber--;
                        }
                    }
                }
                else {
                    cout<<"\nInput Alphabet Error! - Range. Press any key to quit.";
                    getch();
                    return 0;
                }
            }
        }
        if(check_duplicates(temp,temp[alphabetnumber],alphabetnumber)>=0) {
            alphabetnumber--;
        }
        if(alphabetnumber<0) alphabetnumber=0;
        else alphabetnumber++;
        temp[alphabetnumber]=fgetc(fread);
    }
    HERE2:
    *alphabetlist = new char[alphabetnumber];
    if(alphabetlist==NULL) {
        cout<<"\nOut of Memory Error! Press any key to quit.";
        getch();
        return 0;
    }
    for(ictr=0;ictr<alphabetnumber;ictr++) (*alphabetlist)[ictr]=temp[ictr];
    delete temp;
    ictr=checkkeyword(0,"",fread, ';');
    if(ictr==0) return 0;
    return alphabetnumber;
}

//Function: getstates()
//Get state names and accepting state names from fread. Store them in their respective arrays.
int getstates(FILE *fread, int *statectr, int *acceptctr, int *nullctr, string **states,
string **acceptstates, char c)
{
   int ictr, ctr, i1, statenum, acceptstatenum, nfanullstate;
   ictr=ctr=i1=0;
   statenum=*statectr;
   acceptstatenum=*acceptctr;
   nfanullstate=*nullctr;
   char first=0x00, second=0x00;
   string *tempstates, *tempaccept, buffer;
   tempstates= new string[1000];
   if(tempstates==NULL) {
      cout<<"\nOut of Memory Error! Press any key to quit.";
      getch();
      return 0;
   }
   tempaccept= new string[1000];
   if(tempaccept==NULL) {
      cout<<"\nOut of Memory Error! Press any key to quit.";
      getch();
      return 0;
   }
   statenum=acceptstatenum=0;
   for(;c!=';';c=fgetc(fread)){ //main loop
      if(c==EOF){
         cout<<"\nIncomplete File Error! Press any key to quit.";
         getch();
         return 0;
      }
      else if(c==',') { //next state
         second=0x00;
         if(i1==0) {
            cout<<"\nIdentifier Error! - states. Press any key to quit.";
            getch();
            return 0;
         }
         ictr++;
         if(first=='*') { // check to save also in accepting state names
            tempaccept[acceptstatenum]=buffer;
            acceptstatenum++;
         }
         tempstates[statenum]=buffer;
         statenum++;
         first=0x00;
         i1=0;
      }
      else if(c=='*') { // check to see if state being read from file is accepting
         if(i1!=0) {
            cout<<"\nIdentifier Error! - states. Press any key to quit.";
            getch();
            return 0;
         }
         first=c;
      }
      else if(c==' ' || c=='\n' || c=='/') { // ignore white spaces and comments
         if(c=='/') {
               c=fgetc(fread);
               if(c!='/') {
                    cout<<"\nSyntax error after states! Press any key to quit.";
                    getch();
                    return 0;
                }
                comment(fread);
            }
            if(i1>0) second='*';
        }
        else if(check(c)==0)  {
            cout<<"\nIdentifier Error! - Only letters, numbers and underscore are accepted. Press any key to
quit.";
            getch();
            return 0;
        }
        else {
            if(second=='*') {
                cout<<"\nIdentifier Error! - states. Press any key to quit.";
                getch();
                return 0;
            }
            if(i1==0) {
                buffer="";
                if(first=='*');
                buffer+=c;
                i1++;
            }
            else {
                buffer+=c;
                i1++;
            }
        }
    }
    tempstates[statenum]=buffer;
    statenum++;
    nfanullstate=statenum;
    if(first=='*') tempaccept[acceptstatenum]=buffer;
    acceptstatenum++;
    *states=new string[statenum+1];
    *acceptstates=new string[acceptstatenum];
    for(ctr=0;ctr!=statenum;ctr++) {
        if(tempstates[ctr]=="null") {
            cout<<"\nIdentifier Error! Cannot use \"null\" as a state name! Press any key to quit.";
            getch();
            return 0;
        }
        else (*states)[ctr]=tempstates[ctr];
    }
    for(ctr=0;ctr!=acceptstatenum;ctr++) (*acceptstates)[ctr]=tempaccept[ctr];
    delete tempstates;
    delete tempaccept;
    *statectr=statenum;
    *acceptctr=acceptstatenum;
    *nullctr=nfanullstate;
    return 1;
}
```

//Function: gettransition()

//Get NFA transition table and save to transitions[][][][]

int gettransition(FILE *fread, int statenum, int acceptstatenum, int nfanullstate, int alphabetnumber, char *alphabetlist, string *states, string *acceptstates, int ****transition, char c)

```
{
   int ictr, ctr, i1, i;
   string buffer;
   if(c!='{') {
      cout<<"\nIdentifier Error! - transitions. Press any key to quit.";
      getch();
      return 0;
   }
   for(ictr=0;ictr<statenum;ictr++) {
      for(ctr=0;ctr<alphabetnumber;ctr++) {
         for(i1=0;i1<statenum;i1++) (*transition)[ictr][ctr][i1]=-1;
      }
   }
   for(c=fgetc(fread);c!='}';c=fgetc(fread)) {  // main loop
      buffer="";
      for(;c==' ' || c=='\n' || c=='/';c=fgetc(fread)) {
         if(c=='/') {
            if(fgetc(fread)=='/') comment(fread);
            else {
               cout<<"\nTransition Rule Error! Press any key to quit.";
               getch();
               return 0;
            }
         }
      }
      if(c=='}') break;
      for(ictr=0;c!=' ' && c!= ',' && c!='/' && c!='\n';c=fgetc(fread)) {
         if(check(c)==0) {       // get state
            cout<<"\nIdentifier Error! - transitions. Press any key to quit.";
            getch();
            return 0;
         }
         buffer+=c;
      }
      ictr=stateduplicate(states, buffer, statenum);               //check if valid state
      if(ictr<0) {
         cout<<"\nCannot Find State! - transitions. Press any key to quit.";
         getch();
         return 0;
      }
      for(;c==' ' || c=='\n' || c=='/';c=fgetc(fread)) {
         if(c=='/') {
            if(fgetc(fread)=='/') comment(fread);
            else {
               cout<<"\nTransition Rule Error! Press any key to quit.";
               getch();
               return 0;
            }
         }
      }
      if(c!=',') {              //next identifier
         cout<<"\nIdentifier Error! - transitions. Press any key to quit.";
```

```
    getch();
    return 0;
}
for(c=fgetc(fread);c==' ' || c=='\n' || c=='/';c=fgetc(fread)) {
    if(c=='/') {               // get input symbol
        c=fgetc(fread);
        if(c=='/') comment(fread);
        else if(c==' ' || c=='\n') {
            c='/';
            break;
        }
        else if(c==':') {
            c='/';
            ctr=check_duplicates(alphabetlist,c,alphabetnumber);
            if(ctr<0) {
                cout<<"\nCannot Find Input Symbol! - transitions. Press any key to quit.";
                getch();
                return 0;
            }
            goto HERE;
        }
        else {
            cout<<"\nTransition Rule Error! Press any key to quit.";
            getch();
            return 0;
        }
    }
}
if(c=='\\') {
    c=fgetc(fread);
    if(c=='s') c = ' ';
    else if(c==']') c = ']';
    else if(c=='[') c = '[';
    else if(c=='|') c = '|';
    else if(c==';') c = ';';
    else if(c==':') c = ':';
    else if(c=='{') c = '{';
    else if(c=='}') c = '}';
    else if(c=='\\') c = '\\';
    else if(c=='\"') c = '\"';
    else if(c=='/') c= '/';
    else {
        cout<<"\nTransition Alphabet Error! Press any key to quit.";
        getch();
        return 0;
    }
}
ctr=check_duplicates(alphabetlist,c,alphabetnumber);
if(ctr<0) {
    cout<<"\nCannot Find Input Symbol! - transitions. Press any key to quit.";
    getch();
    return 0;
}
for(c=fgetc(fread);c==' ' || c=='\n' || c=='/';c=fgetc(fread)) {
    if(c=='/') {
        if(fgetc(fread)=='/') {
            comment(fread);
        }
        else {
            cout<<"\nTransition Rule Error! Press any key to quit.";
            getch();
            return 0;
        }
    }
}
if(c!=':') {
    cout<<"\nIdentifier Error! - transitions. Press any key to quit.";
    getch();
    return 0;
}
HERE:
for(c=fgetc(fread);c==' ' || c=='\n' || c=='/';c=fgetc(fread)) {
    if(c=='/') {
        if(fgetc(fread)=='/') {
            comment(fread);
        }
        else {
            cout<<"\nTransition Rule Error! Press any key to quit.";
            getch();
            return 0;
        }
    }
}
i=i1=0;
buffer="";
for(;c!=';';c=fgetc(fread)){ // loop to get transition states
    if(c==EOF){
        cout<<"\nIncomplete File Error! Press any key to quit.";
        getch();
        return 0;
    }
    else if(c==',') {
        (*transition)[ictr][ctr][i1]=stateduplicate(states,buffer, statenum);
        if((*transition)[ictr][ctr][i1]<0) {
            cout<<"\nCannot find state! - transitions. Press any key to quit.";
            getch();
            return 0;
        }
        if(i==0) {
            cout<<"\nIdentifier Error! - states. Press any key to quit.";
            getch();
            return 0;
        }
        i=0;
        i1++;
        buffer="";
    }
    else if(c==' ' || c=='\n' || c=='/') {
        if(c=='/') {
                c=fgetc(fread);
                if(c!='/') {
                    cout<<"\nIdentifier Error! - states. Press any key to quit.";
                    getch();
                    return 0;
                    }
                    comment(fread);
                }
            }
            else if(check(c)==0)  {
                cout<<"\nIdentifier Error! - Only letters, numbers and underscore are accepted. Press any key to
quit.";
                getch();
                return 0;
            }

            else {
                buffer+=c;
                i++;
            }
        }
        (*transition)[ictr][ctr][i1]=stateduplicate(states,buffer, statenum);
        if((*transition)[ictr][ctr][i1]<0) {
            cout<<"\nCannot find state! - transitions. Press any key to quit.";
            getch();
            return 0;
        }
        if(i==0) {
            cout<<"\nIdentifier Error! - states. Press any key to quit.";
            getch();
            return 0;
        }
    }
    for(ictr=0;ictr<statenum;ictr++) {
        for(ctr=0;ctr<alphabetnumber;ctr++) {
            for(i1=0;i1<statenum;i1++) {
                if((*transition)[ictr][ctr][i1]==-1) (*transition)[ictr][ctr][i1]=nfanullstate;
            }
        }
    }
    for(ictr=0;ictr<statenum;ictr++) {
        for(ctr=0;ctr<alphabetnumber;ctr++) {
            sorttransition((*transition)[ictr][ctr],statenum);
        }
    }
}

//Function: inittransit()
//initialize transitions[][][][]
int inittransit(int ****transition, int statenum, int alphabetnumber)
{
    int ictr, ctr, i1;
    *transition = (int***)malloc(statenum*sizeof(int**));
    if(transition==NULL) {
        cout<<"\nOut of memory error! Press any key to quit.";
        getch();
        return 0;
    }
    for(ictr=0;ictr<statenum;ictr++) {
        (*transition)[ictr] = (int**)malloc(alphabetnumber*sizeof(int*));
        if((*transition)[ictr]==NULL) {
            cout<<"\nOut of memory error! Press any key to quit.";
            getch();
            return 0;
        }
        for(ctr=0;ctr<alphabetnumber;ctr++) {
            (*transition)[ictr][ctr]=(int*)malloc(statenum*sizeof(int));
            if(*transition==NULL) {
                cout<<"\nOut of memory error! Press any key to quit.";
                getch();
                return 0;
            }
        }
    }
    for(ictr=0;ictr<statenum;ictr++) {
        for(ctr=0;ctr<alphabetnumber;ctr++) {
            for(i1=0;i1<statenum;i1++) (*transition)[ictr][ctr][i1]=-1;
        }
    }
    return 1;
}

//Function: fileoutput()
//Write to output file all necessary data following file format
int fileoutput(char *fname, string name, string *complete, string *completeaccept, string **completetransition, int *passflag, int passnum, int alphabetnumber, int completenum, char *alphabetlist)
{
    int i, k, ctr, ictr;
    FILE *fwrite = fopen(fname,"w");
    if(fwrite==NULL) {
        cout<<"\nError! Cannot open file! Press any key to quit.";
        getch();
        return 0;
    }
    fprintf(fwrite,"FASPEC\nfa ");
    fprintf(fwrite,"%s;\n",name.c_str());
    fprintf(fwrite,"type DFA;\n");
    fprintf(fwrite,"input_alphabet [");
    //alphabet
    for(i=0;i<alphabetnumber;i++) {
        if(alphabetlist[i]==' ') fprintf(fwrite," \\s ");
        else if(alphabetlist[i]==']') fprintf(fwrite," \\] ");
        else if(alphabetlist[i]=='[') fprintf(fwrite," \\[ ");
        else if(alphabetlist[i]=='|') fprintf(fwrite," \\| ");
        else if(alphabetlist[i]==';') fprintf(fwrite," \\; ");
        else if(alphabetlist[i]==':') fprintf(fwrite," \\: ");
        else if(alphabetlist[i]=='{') fprintf(fwrite," \\{ ");
        else if(alphabetlist[i]=='}') fprintf(fwrite," \\} ");
        else if(alphabetlist[i]=='\\') fprintf(fwrite," \\\\ ");
        else if(alphabetlist[i]=='\"') fprintf(fwrite," \\\" ");
        else if(alphabetlist[i]=='/') fprintf(fwrite,"\\/ ");
        else fprintf(fwrite," %c ",alphabetlist[i]);
    }
    fprintf(fwrite,"];\n");
    //states
    fprintf(fwrite, "states ");
    k=0;
```

```
  for(i=1;i<completenum;i++) {
   if(passflag[i]==1) {
    if(k+1==passnum) {
       if(stateduplicate(completeaccept,complete[i],completenum)>=0) {
          fprintf(fwrite," *%s ",complete[i].c_str());
       }
       else {
          fprintf(fwrite," %s ",complete[i].c_str());
       }
    }
    else {
       k++;
       if(stateduplicate(completeaccept,complete[i],completenum)>=0) {
          fprintf(fwrite," *%s, ",complete[i].c_str());
       }
       else {
          fprintf(fwrite," %s, ",complete[i].c_str());
       }
    }
   }
  }
  if(passflag[0]==1) fprintf(fwrite," %s ",complete[0].c_str());
  fprintf(fwrite,";\n");
  //transitions
  fprintf(fwrite,"transitions\n{\n");
  for(ictr=0;ictr<completenum;ictr++) {
     for(ctr=0;ctr<alphabetnumber;ctr++) {
        if(passflag[ictr]==1){
           fprintf(fwrite,"%s , ",complete[ictr].c_str());
           if(alphabetlist[ctr]==' ') fprintf(fwrite," \\s ");
           else if(alphabetlist[ctr]==']') fprintf(fwrite,"\\] ");
           else if(alphabetlist[ctr]=='[') fprintf(fwrite,"\\[ ");
           else if(alphabetlist[ctr]=='|') fprintf(fwrite,"\\| ");
           else if(alphabetlist[ctr]==';') fprintf(fwrite,"\\; ");
           else if(alphabetlist[ctr]==':') fprintf(fwrite,"\\: ");
           else if(alphabetlist[ctr]=='{') fprintf(fwrite,"\\{ ");
           else if(alphabetlist[ctr]=='}') fprintf(fwrite,"\\} ");
           else if(alphabetlist[ctr]=='\\') fprintf(fwrite,"\\\\ ");
           else if(alphabetlist[ctr]=='\"') fprintf(fwrite,"\\\" ");
           else if(alphabetlist[ctr]=='/') fprintf(fwrite,"\\/ ");
           else fprintf(fwrite,"%c ",alphabetlist[ctr]);
           fprintf(fwrite,": %s ;\n",completetransition[ictr][ctr].c_str());
        }
     }
  }
  fprintf(fwrite,"}\nfa_end;");
  fclose(fwrite);
  return 1;
}
```

//Function: convert()
//Complete subset construction. Save in complete[] and completetransition[][]

int convert(string **complete, string **completeaccept, string ***completetransition, string *states, string *acceptstates, int **passflag, int *passtemp, int *completetemp, int statenum, int alphabetnumber, int nfanullstate, int acceptstatenum, int ***transition, char *alphabetlist)

```
{
   int completenum=1, temp2,ictr,ctr,i,j,k,l,acceptindicator,charnum,passnum;
   temp2=ictr=ctr=i=j=k=l=acceptindicator=0;
   string buffer,buffer2="";
   buffer="";
   for(i=0;i<statenum;i++) completenum*=2;
   *complete=new string[completenum];
   *completeaccept=new string[completenum];
   charnum=(completenum*(statenum))+completenum;
   string *bincode;
   bincode=new string[charnum];
   //generate binary code
   for(i=0;i<completenum;i++) {
      l=i;
      j=i;
      for(k=0;k<statenum;k++) {
         l=l/2;
         j=j%2;
         if (j==0) bincode[ctr]="0";
         else bincode[ctr]="1";
         ctr++;
         j=l;
      }
      bincode[ctr]=" ";
      ctr++;
   }
   int temparray[alphabetnumber][completenum], tempnumber[alphabetnumber]; //temporary storage
   for(i=0;i<alphabetnumber;i++) {
      for(j=0;j<completenum;j++) temparray[i][j]=nfanullstate;
   }
   for(i=0;i<alphabetnumber;i++) tempnumber[i]=0;
   j=i=k=l=0;
   *completetransition=new string*[completenum];
   for(ctr=0;ctr<completenum;ctr++) (*completetransition)[ctr]=new string[alphabetnumber];
   //generate state names and transitions
   for(ctr=0;ctr<charnum;ctr++) {
      if (bincode[ctr]==" ") {
         //generate state name and transition after every binary code
         if (buffer=="") {
            buffer="null";
            (*complete)[j]=buffer;
            for(l=0;l<alphabetnumber;l++) {
               (*completetransition)[j][l]= "null";
            }
         }
         else{
         (*complete)[j]=buffer; //buffer store accumulated state names
         for(l=0;l<alphabetnumber;l++) {
            sorttransition(temparray[l],completenum);
         }
         for(ictr=0;ictr<alphabetnumber;ictr++) {
            buffer2=""; //buffer2 will contain accumulated transition state names
            for(temp2=0;temp2<statenum;temp2++) {
               if (temparray[ictr][temp2]<nfanullstate && temp2<statenum) {
                  if (buffer2!="") buffer2+="_";
                  buffer2+=states[temparray[ictr][temp2]];
               }
```

```
                }
                if (buffer2=="") buffer2="null";
                (*completetransition)[j][ictr]=buffer2;
            }
            buffer2="";
            if (acceptindicator=='*') {
                (*completeaccept)[k]=buffer;
                k++;
            }
            }
            acceptindicator=' ';
            j++;
            for(i=0;i<alphabetnumber;i++) tempnumber[i]=0;
            buffer="";
            i=0;
        }
        else { //accumulate state names and transition state names
            if (bincode[ctr]=="1") {
                if (buffer!="") buffer+="_";
                buffer+=states[i];
                for(ictr=0;ictr<alphabetnumber;ictr++) {
                    for(temp2=0;temp2<statenum-1;temp2++) {
                        temparray[ictr][tempnumber[ictr]]=transition[i][ictr][temp2];
                        tempnumber[ictr]++;
                    }
                }
                if (stateduplicate(acceptstates,states[i],acceptstatenum)>=0) acceptindicator='*';
            }
            i++;
        }
    }
    *passflag=new int[completenum];
    passnum=0;
    for(i=0;i<completenum;i++) (*passflag)[i]=0;
    (*passflag)[1]=1;
    passnum++;
    //marking the reachable states
    for(i=0;i<completenum;i++) {
        for(j=0;j<completenum;j++) {
            for(k=0;k<alphabetnumber;k++) {
                if((*passflag)[j]==1) {
                    l=stateduplicate(*complete,(*completetransition)[j][k],completenum);
                    if((*passflag)[l]==1);
                    else passnum++;
                    (*passflag)[l]=1;
                }
            }
        }
    }
    *passtemp=passnum;
    *completetemp=completenum;
    return 1;
}
```

Appendix 3: User’s Manual

A3.1 How to Compile the Source Code?

1. If the computer does not have the Bloodshed Dev-C++ compiler installed, the user can opt to download the Bloodshed Dev-C++ 4.9.9.0 installer from the Bloodshed website at *http://www.bloodshed.net* or use the installer in the CD.
2. Run the installer and follow the instructions to install the compiler.
3. Copy *NFA2DFA.cpp* from the CD to a directory in the hard disk.
4. Run Dev-C++ and open *NFA2DFA.cpp* from the hard disk and press the Ctrl and F9 keys simultaneously. This will compile the source code and at the same time save the source code. *NFA2DFA*.exe will be saved in the same location as the source code.

A3.2 How to Use the Program?

1. Choose or create a directory where *NFA2DFA.exe* will be saved.
2. Save the NFA input file in the same directory.
3. Run the MS-DOS Prompt.
4. Go to the directory where *NFA2DFA.exe* was saved
5. On the command line, type: *NFA2DFA <NFA filename> <DFA filename>* and press <Enter>.
6. The message, "NFA to DFA conversion complete! Press <enter> to quit." will

appear on the screen if the conversion is successful. Otherwise, an error message will be shown. If an error occurs, refer to A3.4.

7. Open the DFA file using Notepad or any text editor to see the results.

A3.3 Some Notes

1. The program does not accept epsilon transitions.

2. Strictly follow the text file format specified in Appendix 1.

3. There is no limit to the size of the alphabet and states that can be used. However, the alphabet is limited to printable characters and the space. It is also important to note that the number of states should not be too large because this takes so much memory space and program run time that might cause the computer to hang.

4. The word "null" can't be used as a state name. The uppercase version (NULL), however, can be used.

5. If a state is duplicated and one is accepting and the other is non-accepting, the program will just consider the first one, whether accepting or not. For instance, if the user entered:

```
states q0, q1, q2, *q1;
```

the program will interpret q1 as a non-accepting state. If it's the other way around:

```
states q0, *q1, q2, q1;
```

the program will interpret q1 as an accepting state.

6. The user will be notified when there are errors in the input file. Once the program has seen an error, it will stop reading the file. Therefore, it won't be able

to detect all the errors in the input file in one run.

7. Comments and white spaces are accepted anywhere in the program (after keyword or identifier, between alphabet, between state names, etc.).

8. If the input file is already a DFA, except that the file type is NFA, the output will be the same file.

A3.4 What Went Wrong?

1. Syntax error after <keyword>
    a. Incorrect syntax after keyword. Refer to the text file format in Appendix 1.
    b. Example:

```
FASPEC
fa /example
myNFA;
typenfa;
input_alphabet [01];
states A,B,C,*D;
transitions
{
A, 0: A, B;
A, 1: A;
B, 0: C;
B, 1: C;
C, 0: D;
D, 0: D;
D, 1: D;
}
fa_end;
```

← This line has a syntax error because comments should begin with a double slash.

← There is an error in this line because the keywords (in this example, 'type') and their corresponding identifier/s should be separated by at least one white space.

2. Keyword Error - <keyword>

    a. Incorrect keyword (spelling or case) or incorrect syntax for comment when a keyword is expected.

    b. Example:

```
FASPEC
la myNFA; //example
type nfa;
input_alphabet [01];
/comment
states A,B,C,*D;
transitions
{
A, 0: A, B;
A, 1: A;
B, 0: C;
B, 1: C;
C, 0: D;
D, 0: D;
D, 1: D;
}
fa_end;
```

← There is an error in this line because the program expects to read ‘fa’, not ‘la’.

← Comments should start with a double slash. This is a keyword error because the program expects to read the keyword ‘states’.

3. Identifier Error - <keyword>

    a. Incorrect syntax of identifier near keyword.

    b. Example:

```
FASPEC
fa myNFA //example
type nfa;
input_alphabet [01];
states A,B,C,N*D;
transitions
{
A 0: A, B;
A, 1: A;
B, 0 C;
B, 1: C;
C, 0: D;
D, 0: D;
D, 1: D;
}
fa_end;
```

← This line has an error because the program expects a semicolon.

← This line has an error because the asterisk should come before the state name.

← This line has an error because there is no comma to separate the state name from the input alphabet.

← This line has an error because there is no colon to separate the input alphabet from the transition state list.

4. End of file error

    a. Incomplete input file. Refer to text file format in Appendix 1.

    b. Example:

```
FASPEC
fa myNFA; //example
type nfa;
input_alphabet [01];
states A,B,C,D;
transitions
{
```

← This is an error because the file is incomplete.

5. Error! You have to enter two filenames.

    a. Program needs two parameters: NFA input filename and DFA output filename.

    b. Example:

```
c:\mydir\nfa2dfa mynfa.txt
```

← There is an error here because the program expects to see another argument for the output DFA filename.

6. Error! Cannot open file.

    a. Program is unable to open NFA input file or DFA output file.

    b. This happens when the NFA file is corrupted or does not exist, or when the DFA output file is read-only.

7. Identifier error! Only letters, numbers and underscore are accepted.

    a. There is a non-acceptable character in an identifier.

    b. Example:

```
FASPEC
fa my-NFA; //example
type nfa;
input_alphabet [01];
states A,B,C,D;
transitions
{
A, 0: A, B;
A, 1: A;
B, 0: C;
B, 1: C;
C, 0: D;
D, 0: D;
D, 1: D;
}
fa_end;
```

← There is an error in this line because identifiers should only consist of letters, numbers or underscore.

8. Identifier Error! - NFA type only.
    a. Program expects to see “nfa” (can be in any case) after the keyword type.
    b. Example:

```
FASPEC
fa myNFA; //example
type dfa;
input_alphabet [01];
states A,B,C,D;
transitions
{
A, 0: A, B;
A, 1: A;
B, 0: C;
B, 1: C;
C, 0: D;
D, 0: D;
D, 1: D;
}
fa_end;
```

← There is an error in this line because the automaton type specified in this file is not an NFA.

9. Out of Memory Error!
    a. Not enough memory to accommodate all alphabets, states, or transitions.
    b. This happens when the computer fails to allocate enough memory for the program to function.

10. Input Alphabet Error! - Non-printable character.

   a. Alphabet can only consist of printable characters.

   b. Example:

```
FASPEC
fa myNFA; //example
type nfa;
input_alphabet [01☺];
states A,B,C,D;
transitions
{
A, 0: A, B;
A, 1: A;
B, 0: C;
B, 1: C;
C, 0: D;
D, 0: D;
D, 1: D;
}
fa_end;
```

← There is an error in this line because ☺ is not part of the allowable characters for the input alphabet.

11. Input Alphabet Error! - Range.

   a. Incorrect use of the dash character to signify range for input alphabet.

   b. Example:

```
FASPEC
fa myNFA; //example
type nfa;
input_alphabet [0-0 0-a a-A Z-A $-#];
states A,B,C,D;
transitions
{
A, 0: A, B;
A, 1: A;
B, 0: C;
B, 1: C;
C, 0: D;
D, 0: D;
D, 1: D;
}
fa_end;
```

← There are four errors in this line because the dash (to specify a range) was improperly used. Only the range 0 to a is correct.

12. Identifier Error! Cannot use "null" as a state name!

    a. The string “null” cannot be used as a state name. The state name “null” is used exclusively for the output DFA. The user, however, can use the state name “NULL” or any other variation in case (e.g. NuLl) as long as not all letters are in the lower case.

13. Transition Rule Error!

    a. Incorrect syntax for the comment inside the transitions section.

    b. Example:

```
FASPEC
fa myNFA; //example
type nfa;
input_alphabet [01];
states A,B,C,D;
transitions
{
A, /comment
0: A, B;
A, 1: A;
B, 0: C;
B, 1: C;
C, 0: D;
D, 0: D;
D, 1: D;
}
fa_end;
```

← There is an error in this line because comments should start with a double slash.

14. Cannot Find State! - transitions.

   a. State used in the list of transition rules is not part of the state list.

   b. Example:

```
FASPEC
fa myNFA; //example
type nfa;
input_alphabet [01];
states A,B,C,D;
transitions
{
A, 0: E, B;
A, 1: A;
B, 0: C;
B, 1: C;
C, 0: D;
D, 0: D;
D, 1: D;
}
fa_end;
```

← There is an error in this line because "E" is not declared in the state list.

15. Transition Alphabet Error!

   a. Incorrect use of the escape sequence in the transitions section for the input alphabet.

   b. Example:

```
FASPEC
fa myNFA; //example
type nfa;
input_alphabet [01];
states A,B,C,D;
transitions
{
A, 0: A, B;
A, \1: A;
B, 0: C;
B, 1: C;
C, 0: D;
D, 0: D;
D, 1: D;
}
fa_end;
```

← There is an error in this line because the escape sequences, started by the backslash, do not include 1.

16. Cannot Find Input Symbol! - transitions.

   a. Character used in the list of transition rules is not part of the input alphabet list.

   b. Example:

```
FASPEC
fa myNFA; //example
type nfa;
input_alphabet [01];
states A,B,C,D;
transitions
{
A, a: A, B;
A, 1: A;
B, 0: C;
B, 1: C;
C, 0: D;
D, 0: D;
D, 1: D;
}
fa_end;
```

← There is an error in this line because 'a' is not declared in the input alphabet.

References

CE 160 Website. *http://www.geocities.com/lui_agustin/ce160*.

Hopcroft, et al. 2001. Introduction to Automata Theory, Languages, and Computation. Singapore: Pearson Education Asia Pte Ltd.