Alexandria University
Faculty of Engineering
Computer and System Engineering
C321: Programming languages and compiler

Syntax Directed Translation Scheme

Compiler Project

phase1

—

Names: **ID:**

- Afnan Mousa   15
- Enas Morsy   20
- Sara Mohammad   31
- Shimaa Kamal   33

Non-Deterministic Finite Automata

Data structures:

- ❏ struct Transition :
    - ● struct state* next
    - ● string input_symbol
- ❏ struct copied_state :
    - ● state * related .
- ❏ struct state :
    - ● int id
    - ● copied_state cpd
    - ● bool accepted = 0   // 0-> not accepted state   1-> accepted state
    - ● string accepted_language=""   // accepted Pattern name
    - ● int priority=0   // priority of the accepted pattern
    - ● vector<Transition> transitions
- ❏ struct automata :
    - ● state* start
    - ● state* end_
- ❏ Vector < pair < string,automata > > Languages :
    - ★ Contains the definitions (the name and their automatas) .
- ❏ vector < pair < string,automata > > patterns :
    - ★ contains the expressions (the name and their automatas) .
- ❏ vector < string > keyWords :
    - ★ contains the keywords .
- ❏ vector < char > punctuation :
    - ★ contains the punctuations .

Functions:

To Calculate the NFA automata 4 classes is used :

- ❏ Input_rules :
    - ● Int main () : it reads the **rules.txt** file and sends line by line to the extract() .
    - ● Void extract (string line) : it checks the line
        - ➢ If it contains " **=** " without "**/** " before it and doesn't contain " **:** "
            - ➔ It's a definition .
            - ➔ Remove the extra spaces from the line .
            - ➔ Split the line using " **=** " to get the name and regex parts .

➔ Call language_NFA () from calc_rgx using the regex part from line to calculate its automata .
➔ Store the automata and the name part in the **Languages** vector .

➢ If it contains " **:** "
➔ It's an expression .
➔ Remove the extra spaces from the line .
➔ Split the line using " **:** " to get the name and regex parts .
➔ Call language_NFA () from calc_rgx using the regex part from line to calculate its automata .
➔ Store the automata and the name part in the **patterns** vector .

➢ If It starts with " **{** " :
➔ split the line with space .
➔ Push the parts in the **keyWords** vector .

➢ If it starts with " **[** " :
➔ split the line with space
➔ push the parts in the **punctuation** vector .

❏ Parse_rules :
➢ bool is_closure(char c) :
➔ returns true if the character is '*' or '+' or '.'
➢ bool is_AndOr(char c) :
➔ returns true if the character is '|' or ' '
➢ bool split_space (int i, string str) :
➔ returns true if the space in index i is between 2 letters .
➢ String removeSpaces(string str) :
➔ to remove any space except that applies split_space() .
➢ String removeExtraSpaces(string str) :

" **not to remove space is concatenation** "

➔ To keep the space only if
★ it's after ") " .
★ It's before " (" but if the "(" is the first character in the regex .
★ it's between (E or L) and (not is_AndOr() character) .
★ It's between (is_closure()) and (not is_AndOr() character) .
➢ vector<string> splits(string str, string del) :
➔ Split the string using substring into parts .

❏ calc_rgx :
➢ int find_language(string lang,vector<pair<string,automata> > Languages) :
➔ Check if the Languages vector contains the lang , return its index .
➔ Otherwise return -1 .
➢ Bool is_operator(char c) :

- ➔ Returns true if c equals ' ' or '|' or '*' or '+ ' .
- ➢ int :priority(char op)
  - ➔ Returns the precedence of the operators . from highest Closure >> Concatenation >> OR .
- ➢ void dfs(state* st, state* cpd, state* helper_end)
  - ➔ Traverse automata graph using dfs algorithm to Copy all the states and transitions .
- ➢ automata start_copy(automata n)
  - ➔ Copy an automata using dfs()
- ➢ automata applyOp(automata* a, automata* b,char op, state* s, state* e)
  - ➔ Check the op character and call the suitable function from **NFA** class.
- ➢ automata language_NFA (string rgx,vector<pair<string,automata> > Languages)
  - ➔ Declare 2 stacks one of character to store operators and one of automatas .
  - ➔ If the current character is " (" , push it to the ops stack .
  - ➔ If the current character is ") " , pop ops until the " (" is found and for each operator pop from the automatas stack apply the operator and push the result in the automatas stack .
  - ➔ If the current character is operator , check the priority of the top of ops () if greater than or equal the current operator apply the pop the top of ops and apply it then push result to automatas stack.

    Push the current character to ops stack .

  - ➔ Else read all the characters as one token , look for it in the Languages vector if found, take copy from its automata and push it on the automatas stack and if not found construct its automata and push it .
- ➢ automata combine (vector <pair<string,automata>> patterns,state* s)
  - ➔ Calls combined_NFA() from NFA class passing patterns and s .
- ❏ NFA :
  - ➢ automata basic_NFA (string a, automata base)
    - ➔ Construct basic automata with only 2 states and transition between them under input **a .**
    - ➔ push the transition to the start state transitions vector .
  - ➢ automata and_NFA (automata* automata1, automata* automata2)
    - ➔ Concatenate 2 automatas by adding a transition from the **first automata end** state to the **second automata start** state under input " \L" .
    - ➔ push this transition to the first automata end state transitions vector .
    - ➔ Returns the result automata .

- ➢ automata or_NFA (automata* automata1, automata* automata2, state* new_start, state* new_end)
  - ➔ Add 1 to the states number and set it as id to the new_start state .
  - ➔ Add 1 to the states number and set it as id to the new_end state .
  - ➔ Adding a transitions :
    - ★ From the **new_start** state to the **start of the first automata** state under input " \L"
    - ★ From the **new_start** state to the **start of the second automata** state under input "\L"
    - ★ From the **end of the first automata** state to the **new_end** state under input "\L"
    - ★ From the **end of the second automata** state to the **new_end** state under input "\L"
  - ➔ Returns the result automata .
- ➢ Automata Closure(automata * automata1, state* new_start, state* new_end, char c) :
  - ➔ Add 1 to the states number and set it as id to the new_start state .
  - ➔ Add 1 to the states number and set it as id to the new_end state .
  - ➔ Adding a Transitions :
    - ★ From the **end state of the automata** to the **start state of the automata** under input "\L"
    - ★ From the **new_start** state to the **start state of the automata** under the input "\L"
    - ★ From the **end state of the automata** to the **new_end** under input "L "
    - ★ Only if c equals "*", From the **new_start state** to the **new_end state** under input "\L" .
- ➢ automata combined_NFA(vector <pair<string,automata>> patterns, state* new_start) :
  - ➔ Combine all the patterns in one NFA automata .
  - ➔ Add 1 to the states number and set it as id to the new_start state .
  - ➔ Loop on the patterns vector .
  - ➔ For each pattern automata set the end state the accepted equals to 1 , accepted Language equals the pattern name and the priority equals the pattern index .
  - ➔ Set a transition from the **new_start state** to **each start state of the patterns** under input "\L" .

Algorithm:

- ❏ Read the **rules.txt** file line by line .
- ❏ Call the **extract()** from **input_rules** class .

- ❏ **extract()** function splits the regex part and calls **language_NFA()** in **calc_rgx** with it .
- ❏ **language_NFA()** construct automata using **NFA** methods .
- ❏ Store the automata in **Languages** or **patterns** vectors .
- ❏ After finishing all the lines , combine them in the final NFA automata using **combine()** in **cal_rgx** .

Assumptions:

★ The rules are written correctly .

Deterministic Finite Automata

Data structures :

➢ The main Data structures use to convert from **<u>Nondeterministic finite automata</u>** to **<u>Deterministic finite automata</u>** are :
- ❏ **<u>struct DFA_State</u>** contains attributes which represent each node of the DFA graph .
  - int id
  - bool accept_state_flag = false //0 not accepting ,1 for accepting.
  - string name=""
  - int priority = 0
  - set <state*> subset
  - set <int> subset_ids
  - map <string, set<state*>> symbols
  - map <string, set<int>> Group_ids
- ❏ **<u>struct DFA_Graph</u>** it represents each single state in the DFA automata where it contains:
  - bool acceptance_state: indicates that whether this state is an accepting one or not
  - string name: contains the name of the language or pattern this state belongs to if it is an accepting one.
  - int priority: the priority of the state language
  - map<string, int> : contains all possible inputs this state can get and the corresponding states' ids for each one

Functions:

➢ To convert from the NFA to the DFA using the **class "DFA"** which contain basic function to do that are :

- ❏ map<int, DFA_Graph> Subset_Construction(state* original)
  - ➜ Take the start node of the final automate result from the first step of the program .
  - ➜ Create a pointer on the new node from struct DFA_State to represent the **start node of DFA** set the main values as (name, priority, accepted).
  - ➜ Insert the original state which represents the start state in NFA to the **set "subset"** and the id of this node to **set subset_ids** .
  - ➜ Call the function **E_closure** to return a vector of pointer for each node that can reach it by epsilon from the start node .
  - ➜ Insert the content of this vector under the set of **set "subset"** and the ids under **set subset_ids** which belong to the **initial state of DFA** .
  - ➜ Make a vector of DFA_State * represent the **Table of DFA** .
  - ➜ Call the function Move_To for each element in this vector until it reaches the end of the vector.
  - ➜ Call **get_graph function** to represent the final graph of DFA .
  - ➜ Call the **minimize_graph function** to minimize this graph .
  - ➜ Return the graph which Produce after minimization.

- ❏ vector <state*> E_closure(state* original)
  - ➜ Take NFA node _state_ .
  - ➜ Create a stack of a pointer on struct state .
  - ➜ Push the **original state** on it .
  - ➜ Make a while loop until this stack becomes empty .
  - ➜ In this loop first :
    - ● Pop NFA pointer from stack .
    - ● Loop on all transitions of this state .
    - ● If any of the transition occurs under condition epsilon .
    - ● Store the pointer of this node in the vector without repetition.
    - ● Return this vector .

- ❏ void Move_To(DFA_State *Basic_node)
  - ➜ Take the start node of the DFA state .

➔ Make a loop on all NFA nodes which composite under the start node of DFA .
➔ For each node of the NFA make a loop of all transitions to another NFA nodes .
➔ If this transition **is epsilon skip** .
➔ If this transition reaches an accepted state mark the DFA state under specific condition is accepted state.
➔ Store the pointer of all the NFA states reached under specific conditions .
➔ Store although the ids of these NFA states.
➔ Call **E closure** for all NFA states , insert this state as mentioned previously.
➔ In this point we have a map that under input specific string the state changes from the start node of DFA to the next node which represents a group of NFA states represented as a set .
➔ Loop on this map check that the new node didn't exist before then create a new DFA node with all attributes and push it in the **vector Table** .

❏ map<int, DFA_Graph> get_graph()
➔ Create a pointer on the new node of DFA_Graph to represent the start node.
➔ Loop on the Table vector .
➔ set the main values as (id,name, priority, accepted,transition_map).
➔ The transition_map represents under which input the DFA node_state_ goes to the other DFA node_state_ .
➔ Create map the key represents the id of DFAV state and the value is the DFA_Graph node .
➔ After that if we need to print the graph call the function print_graph .
➔ Finally return the main Map .

Algorithm:

➢ The minally algorithm to convert from NFA to DFA depends on the **Subset_Construction technique** which split to two basically function :
  ● first one to get an epsilon state .
  ● another one to represent the transition from node to another .
➢ The main function is called the **Subset_Construction function** in the **class DFA** after building the automate .
➢ The Subset_Construction function is called the **E_closure** function.

➢ Then the Subset_Construction function is called the **Move_To** function.
➢ Then call **get_graph()** function .
➢ Which call **print_graph** function to print the **final table of transitions of the DFA**.

DFA Minimization

Data structures :

❏ It uses **DFA_Graph struct** created in DFA class, that was predefined in DFA section

Functions:

❏ **map<int, DFA_Graph> minimize_graph(map<int, DFA_Graph> graph):**
    ➢ This function receives created DFA in form of **DFA_graph** struct
    ➢ Calls essential minimization functions to modify sent automata
    ➢ Return the minimized automata after modification
❏ **vector<set<int>> table_filling(map<int, DFA_Graph> graph):**
    ➢ This function builds a 2D-matrix of size equals number of DFA states to represent all possible combinations between them
    ➢ It is an implementation of **Myhill**-**Nerode theorem** where after performing essential steps of **Table Filling** technique each entry represents whether the two states indicated by row and column index should be joined as one state or not
        ○ If the entry is 0 that means they are the same state
        ○ If the entry is -1 that means they are separate states
    ➢ It returns a vector of set **equivalent_states** where each entry in the vector is a set of states ids indicating that all states with these ids are equivalent and should be represented using only one state
❏ **void join_states(vector<set<int>> &equivalent_states, int id1, int id2):**
    ➢ This function receives the ids of two equivalent states
    ➢ It first checks if any of them belongs to a specific set in the **equivalent_states** vector, it pushes the other one in the same set
    ➢ Otherwise it creates a new entry in this vector and push new set represented by id1&id2 in this entry
❏ **map<int, DFA_Graph> minimized(map<int, DFA_Graph> graph, vector<set<int>> equivalent_states):**
    ➢ This function represents all equivalent states in the **equivalent_states** vector in the DFA graph using only one state of them, the first state in each set, and removes the rest .
    ➢ Before removing any state it goes through the whole DFA and changes any reference or pointer to this state by another one to the set first state.

➢ It returns DFA after minimization.
❏ **string get_pointer_id(int id, map<int, DFA_Graph> graph):**
 ➢ Most of previous functions deals only with state id not the state itself
 ➢ This function receives the sent id as will as DFA and goes through all the automata states till it find the automata declared by this id
 ➢ It returns this state language or pattern name if it belongs to any.

Algorithm:

❏ The used technique to minimize the DFA is **Table Filling method** which depends on **Myhill**-**Nerode theorem**, where all states combinations represented by entries in a 2D matrix; the row index represents the first state and the column represents the second one
❏ This matrix in initialized by 0 representing that all states can be joined in one state
❏ Then it goes through each state and marks each two states as unequivalent states by making their entry = -1 in the following cases:
 ➢ one of them is accepting state and the second isn't.
 ➢ Both of them are accepting states but don't belong to the same language; don't represent the same pattern
❏ After that it goes through all entries and check the dependencies between them, if two states to be equivalent depending on other two states, it checks whether the others one are equivalent or not and mark the first entry depending on that
❏ It continues in this loop till it reaches a stage where it scans the whole matrix and doesn't apply any change to its entries
❏ After Filling the matrix the program go through it and sends the ids of any two states marked as equivalent to the **join_states** function
❏ **Join_states** builds a vector of sets where every entry represents all equivalent states
❏ The real minimization step is done at **minimized** function where it makes the first state of each equivalent set represent the whole set in the DFA by:
 ➢ Removing all other states in this set from the DFA
 ➢ changing any reference or pointer to any removed state by its set representative.
❏ At this step, DFA is minimized and returned .

Validation and Error Recovery

Data structures:

❏ **map<string, int> next** that saves the next states to the current state. It is a pair of strings that represents the input which makes the current state move to the next

state, and an int that represents the id_number of that next state.

❑ **map<int, DFA_Graph> graph** that represents the final DFA. It uses **DFA_Graph struct** created in DFA class, that was predefined in DFA section

Functions:

❑ **Split (**string test_program**)**
It takes the code from the console and starts splitting it with whitespaces to form the tokens which are supposed to be parsed character by character. And pushing them into a vector called "splitted".

❑ **Parse** (punctuation, keyWords, splitted)
At first it checks each token of the splitted vector to see if it is a keyword or a punctuation.
If not, it passes this token to the "validation" function which will be described below.

❑ **Validation** (graph, string word)

It starts parsing the given word character by character to apply the DFA on it. The whole algorithm of this function will be described briefly in the Algorithm section.

Algorithm:

The main algorithm here is done on the "validation" function. That appends the characters of the passed token character by character and at each character.

➢ It starts from the **start state** of the **DFA** and keeps track of its all next states under different inputs.
➢ Then it loops over the token on each character and checks if its regex representation can match any regex of the current state input.
➢ If so, it keeps track of the last reached state and updates the current state to be its next state under this input.
➢ It also used to implement the **maximal munch**. It checks if this next state is accepted or not. If it is accepted, it keeps track of the last accepted state and the index of the current char on the token.
➢ Else if the character can't be an input for this current state, it splits the token to a new token that started from the index of the string that has the last accepted state. And **recursively** calls the validation function to validate the remaining token.
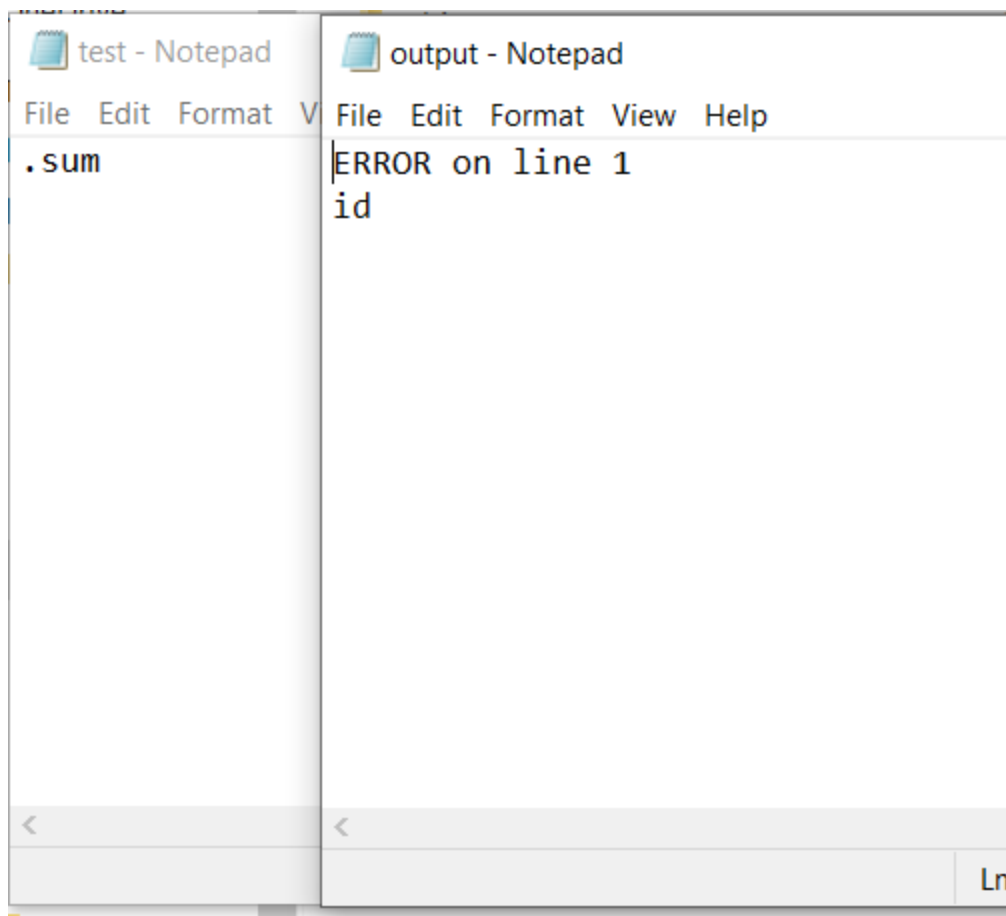
➢ At the end it checks if the last accepted state is the last state that this token reaches, if so, then it accepts the token as a valid token and prints on the output file its name.

For **Error recovery,** when the character input can't be a valid input to any state. We set it as an error.
And to recover from this error, we just delete successive characters from the remaining token and recursively check if we find a valid token by the validation function.

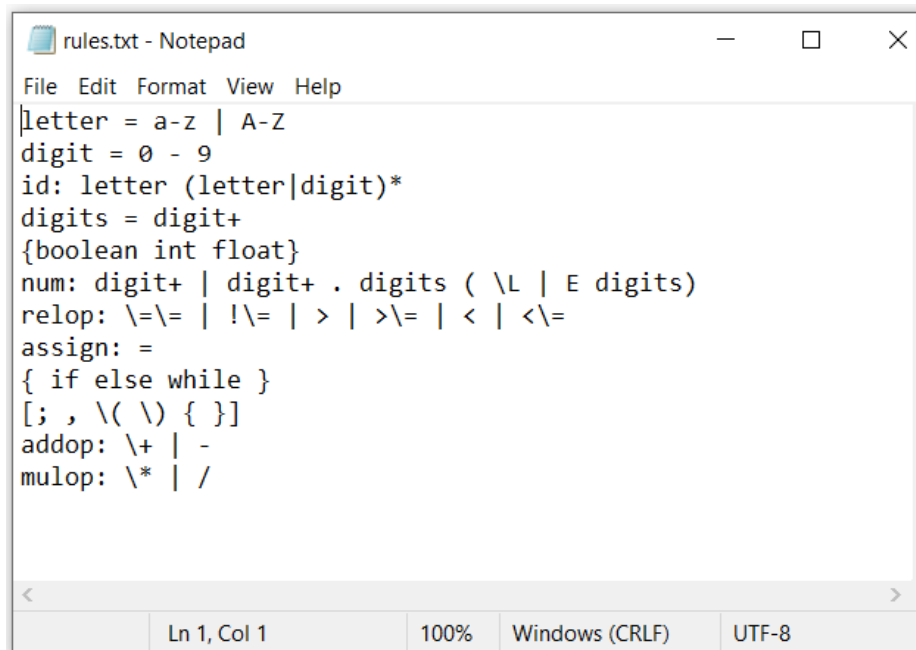Sample run For example for error recovery:
".sum" is not valid. So it prints an error and starts searching for the next valid token which is "sum" as an "id".

test - Notepad

File   Edit   Format   V

.sum

output - Notepad

File   Edit   Format   View   Help

ERROR on line 1
id

Ln

Sample Runs


```
rules.txt - Notepad                                    —    □    ✕
File  Edit  Format  View  Help
letter = a-z | A-Z
digit = 0 - 9
id: letter (letter|digit)*
digits = digit+
{boolean int float}
num: digit+ | digit+ . digits ( \L | E digits)
relop: \=\= | !\= | > | >\= | < | <\=
assign: =
{ if else while }
[; , \( \) { }]
addop: \+ | -
mulop: \* | /




<                                                             >
          Ln 1, Col 1          100%   Windows (CRLF)     UTF-8
```

Transition Table for The Minimal DFA


```
DFA after minimization

 *******************
state id is: 1
its accepting state is 0
under input !   it goes to state 2
under input *   it goes to state 3
under input +   it goes to state 4
under input -   it goes to state 4
under input /   it goes to state 3
under input 0-9   it goes to state 7
under input <   it goes to state 8
under input =   it goes to state 9
under input >   it goes to state 8
under input A-Z   it goes to state 11
under input a-z   it goes to state 11
*******************

state id is: 2
its accepting state is 0
under input =   it goes to state 13
*******************

state id is: 3
its accepting state is 1
*******************
```

```
state id is: 4
its accepting state is 1
*******************

state id is: 7
its accepting state is 1
under input .  it goes to state 14
under input 0-9  it goes to state 7
*******************

state id is: 8
its accepting state is 1
under input =  it goes to state 13
*******************

state id is: 9
its accepting state is 1
under input =  it goes to state 13
*******************

state id is: 11
its accepting state is 1
under input 0-9  it goes to state 11
under input A-Z  it goes to state 11
under input a-z  it goes to state 11
*******************
```

```
state id is: 13
its accepting state is 1
*******************

state id is: 14
its accepting state is 0
under input 0-9  it goes to state 21
*******************

state id is: 21
its accepting state is 1
under input 0-9  it goes to state 21
under input E  it goes to state 22
*******************

state id is: 22
its accepting state is 0
under input 0-9  it goes to state 23
*******************

state id is: 23
its accepting state is 1
under input 0-9  it goes to state 23
*******************
```
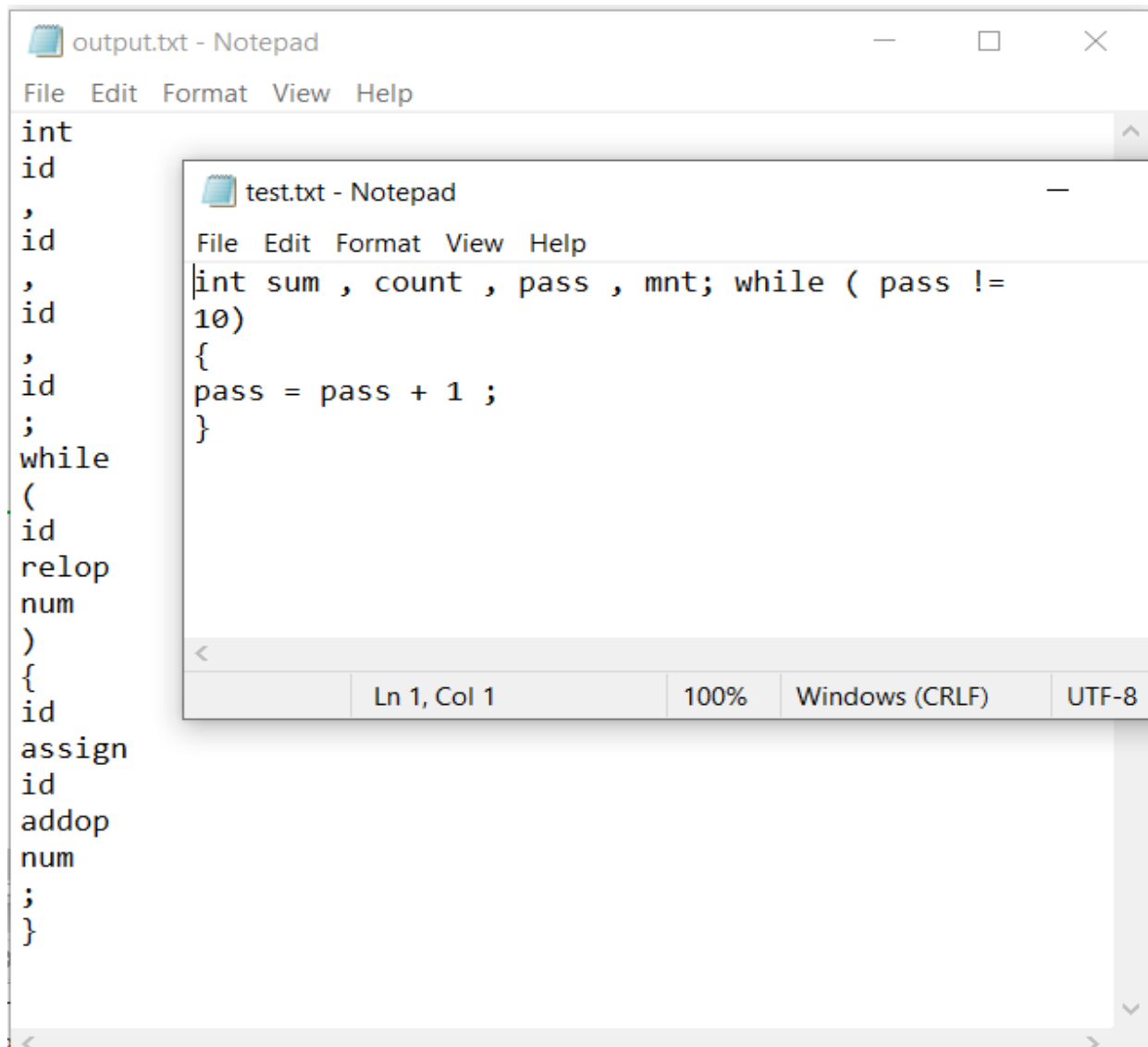
Test Program

➢ Our output file of the test program:

```
output.txt - Notepad
File  Edit  Format  View  Help
int
id
,
id
,
id
,
id
;
while
(
id
relop
num
)
{
id
assign
id
addop
num
;
}
```

```
test.txt - Notepad
File  Edit  Format  View  Help
int sum , count , pass , mnt; while ( pass !=
10)
{
pass = pass + 1 ;
}

Ln 1, Col 1        100%    Windows (CRLF)    UTF-8
```