**CMPS 323 , Handout No.3**

**Assignment No.3**

**OTHER FORMS OF GRAMMARS**

**Notations.**

1. [a] means a or λ. Example. For w=b[a], then either w=bλ=b or w=ba
2. | means or, +, or union. Example: a\*|b\* is the same as a\*+b\*.
3. {a} means any power of a including the zero power. {a} is the same as a\* ,

Examples: (a+b)\*= { a|b } , a\* + b\* = {a} | {b}, and a\*b\* = {a}{b}

1. The ***Syntax diagram*** of the above notations:

|  |  |
| --- | --- |
|  |  |
| Example: **S🡪[a] and its syntax diagram**  S this arrow generates λ    a  a  S🡪[a] or S🡪a | λ  Start at S, the straight line is λ (S=λ). But starting from S if we go down and then go up then S=a | Example: **S🡪{a} and its syntax diagram**  S a0 or λ  This is a loop  this  a  S🡪{a} or S🡪aS | λ  Start at S, the straight line is for S=λ=a0 . Start at S we can go through the loop and go back to the loop before we exit. Hence S=a (go through loop once), S=a2 (go through loop twice),….. |

1. ***The Backus Nour Form (BNF),*** is a CFG when we write one rule per line. Example:

|  |  |
| --- | --- |
| **CFG** | **CFG in BNF form** |
| A🡪aA | bB | λ  B🡪bB | λ | A🡪aA  A🡪bB  B🡪bB  A🡪λ  B🡪λ |

1. ***Extended BNF (EBNF), i***s a CFG in which we write the language of CFG using the notations [ ], { }, and |

|  |  |
| --- | --- |
| **CFG** | **EBNF form of CFG** |
| CFG: A🡪aA | λ a  The FA of this grammar is :    The language of this CFG is L=a\* | It is much easier to find the EBNF from the Language,  For L= a\* , then  CFG: A🡪aA |λ  EBNF: A🡪{ a } |
| CFG: A🡪aA | bA | λ a b  FA:  Language: L=(a+b)\* | L= (a+b)\*  CFG: A🡪aA | bA | λ  EBNF: A🡪{ a | b } |
| E🡪aA |bB | λ a A B b  A🡪aA| λ  B🡪bB | λ a b  FA: E  L=a\* + b\* | L= a\* + b\*  CFG: E🡪aA|bB|λ, A🡪aA|λ, B🡪bB|λ  EBNF:E🡪{a} | {b} |

The following are general examples to cover all cases at the same time.

**Example.** For each language, find its (i)FA,(ii)CFG,(iii)CFG in BNF,(iv) CFG in EBNF, (v) Syntax diagram

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Language** | **FA** | **CFG** | **BNF** | **EBNF** | **Syntax diagram** |
| L=a\* | a  X | X🡪aX | λ | X🡪aX  X🡪λ | Since L=a\*, the  EBNF grammar  is X🡪{a} | X  a  the top line is λ |
| L=a\*b\* | a b  b    A B | A🡪aA|bB|λ  B🡪bB|λ | A🡪aA  A🡪bB  B🡪bB  A🡪λ  B🡪λ | L=a\*b\* implies  A🡪{a}{b} | A  a b |

It is clear that to find the EBND and syntax diagram there is no need to have the FA or the CFG of the language. The following examples are to find EBNF grammar, and the syntax diagram directly from the languages.

**Examples:** Find the CFG of each language in EBNF and syntax diagram

|  |  |  |
| --- | --- | --- |
| **Language (L)** | **EBNF grammar of L** | **Syntax diagram to describe grammar of L** |
| L=aa\* + bb\*  Only powers of a or powers of b  λ is not in L | L = aa\* + bb\*  S🡪a{a} | b{b} | a  A  B  a  S  b  b |
| L=(a+b)\*  ={λ, powers of a, powers of b, any combinations of a’s and b’s} | The \* means { }, and a+b means a|b. Thus  S🡪{ a | b } | S  B  b  a  The top straight line is for λ  The top loop generates all powers of a,  The other loop generates all powers of b,  Combination of the loops generate words made up of a’s and b’s |
| L=(a+b)c\* | S🡪( a| b){c}=a{c} | b{c} | a  S  b  c |
| L=(a+b+c)\* | S🡪{a|b|c} | S  a    b    c |

Note: The following syntax diagrams representing L1=aa\* and L =a\*

|  |  |
| --- | --- |
| L= aa\*  a  S | L = a\*  λ  S  a |

**Examples.**

|  |
| --- |
| 1. Identifiers in C++. An identifier is a string of letters, digits, and underscores. Identifiers must begin with a letter or underscore. Construct syntax diagram and write its EBNF for the CFG of identifiers in C++   Letters(L)  EBNF grammar  <id> <id> 🡪 L | U | LX |UX  L  X 🡪{L | D | U }  D  Underscore(U) L 🡪 a|b|…|z|A|B|….|Z  D -> 0|1|2|……9  U  U 🡪\_ |
| 1. Given EBNF grammar S🡪{a} c [d], Construct its syntax diagram, and write the grammar in form of BNF   S c The BNF of the grammar look like this  S🡪 A c D, where A={a}, D=[d], and c is fixed  a d A🡪aA ,  A🡪 λ  D🡪d  D🡪 λ |
| 1. L =a\*( b\* + c ), construct the syntax diagram of L. Write the grammar of L in EBNF format   b      c  S The EBNF grammar of L is:  S🡪 {a} ( {b} | c )  a c |
| 1. Write the BNF of the following EBNF grammars 2. S🡪{a}{b} ii. S🡪{a|b}   S🡪AB, let T=a|b, then S🡪{T} or S🡪T\*  A🡪aA Hence:  A🡪 λ, S🡪TS  B🡪bB S🡪 λ  B🡪λ T🡪a  T🡪b |
| 1. S🡪{ [a] b } c , construct the syntax diagram of this EBNF grammar, and write the grammar in BNF.   S c The BNF of this grammar is:  b  S 🡪 Xc , wher X={ [a]b}  A X🡪 YX , where Y=[a]b=ab or λb =ab or b  a  Y🡪ab  Y🡪 b  X🡪λ |
| 1. Convert S🡪[a] { b | c } to CFG   Let A=[a] and X={b | c }, then the grammar becomes S🡪AX, where A=a or λ and X=(b+c)\*  Lets do one more substitution: X=(b+c)\* or X=R\* where R =b | c. Now put all pieces  Together to have the BNF format of the given grammar  S🡪AX , where A=[a]=a, λ and X={a|b}=(a+b)\* =R\* , R=a,b. X=R\* or X🡪RX , λ  A🡪a | λ  X🡪RX |λ  R🡪b | c |
| 1. Construct EBNF grammar for simple function headings in C++( for example: void f(), int f(int a, int b).   <functions heading> 🡪 <type> <id> ( [ <type><id>{ , <type><id> } ] )  <type>🡪void| int |float |string  <id> 🡪(<letter>|<underscore>){<letter>|<digit>|<underscore>}  <letter>🡪<upper>|<lower>  <upper>🡪A|B|C|………|Z  <lower>🡪a|b|c|……….|z  <digit>🡪0|1|2|………..|9  <underscore>🡪\_  Trace the grammar for function heading: int sum( int a1, int a2, int a3)  <function heading>  <type> <id> ( [ <type> <id> {, <type <id>} ] )  <type><id> , <type> <id> , <type><id> )  int sum ( int a1 , int a2 , int a3 ) |

**REGULAR AND NON-REGUALR LANGUAGES**

Definition. Language L is regular if we can write L using ( ), \* and +. Otherwise the language is called a ***non-regular*** language.

**Example.**

|  |  |
| --- | --- |
| **Regular languages** | **Non-regular languages** |
| 1. L=a\*b\* 2. L=(a+b)\* 3. L=ab\* + ba\* | L={ anbn |n=1,2,3,…}={ab, aabb, aaabbb,…….}  Notice that this language is not L2=a\*b\*. word aaab is in L2 but is not a member of L |

**TERMINAL AND NON-TERMINAL SYMBOLS**

Consider the following CFG:

A🡪 aA| bB

B🡪 bB | λ

Then Set of Terminals ={a,b,λ } and set of non-terminals = { A,B }. In the other word, ***terminals*** are symbols that are not expandable and ***non-terminals*** are expandable. Suppose we want to use this grammar to trace the word= aab, then : A

/ \

a A

/ \

a A

/ \

b B

a a b λ = w , Terminals={a,b,λ},

non-terminals={A, B}

**REGULAR AND NON-REGULAR CONTEXT-FREE-GRAMMARS**

**Definition.** CFG is ***regular*** if each rule in the grammar is in one of the following forms:

1. < non-terminal> 🡪 string of terminals with **exactly ONE** non-terminal at the END
2. < nonterminal >🡪 terminals including λ

**Example**. For CFG: X🡪aX | bX | λ, in which terminals={X} and non-terminals={a,b, λ}. The first two rules satisfy rule (a) and the last one satisfies rule (b). Therefore this CFG is a regular .

**Definition.** If the CFG is not regular, then it is called a ***non-regular CFG***

**Example.** Given CFG: E🡪AB, A🡪aA | λ, B🡪Bb. The first rule E🡪AB does not satisfy rule (a) above (there are more than one non-terminal ( A,B)on the right-hand-side) therefore this grammar is non-regular

**Recall: FA**

**Regular language Regula CFG**

We have covered all these conversion cases in handout 1 and 2

**CONSTRUCTING REGUALR AND NON-REGULAR CFG FOR A GIVEN REGULAR LANGUAGE**

There are two methods to find a CFG for a given regular language

**Example.**

|  |  |  |
| --- | --- | --- |
| **Regular language** | **Regular CFG**: construct FA and then use it to write regular CFG | **Non-regular CFG**: Write the non-regular CFG directly using the language |
| L=a\*b\* | Construct an FA to accept L  a b  b  A B  Write regular CFG using FA  A🡪aA| bB|λ  B🡪bB|λ | L = a\* b\*  Let A=a\* and B=b\*, then  Write non-regular CFG  S🡪AB , two nonterminal; make it  Non-regular CFG  A🡪aA |λ  B🡪bB|λ |
| L=b\*(a+b)a\*  λ is not in L, thus initial state is not a final state | FA:  b a  a,b  A B  Regular CFG:  A🡪bA |aB | bB  B🡪aB|λ | L = b\*(a+b)a\*  Let B=b\*, X=a+b, and A=a\*. Then  Non-regular CFG look like this  S🡪 BXA  B🡪bB|λ  X🡪a | b  A🡪 aA|λ |

**DETERMINISTIC AND NON-DETERMINSITC FINITE AUTOMATOA**

Consider the following two cases:

|  |  |
| --- | --- |
| Case 1  a  A B  a is accepted by this machine | A Case 2  a a  B C  Start at state A, for input a we have two choices, either enter state B (a is accepted) or enter state C ( a is not accepted). But in handout 1, we said word is accepted by FA if “there is a way” to start at initial state and enter a final state. Hence a is accepted by this machine.  This is an example of a ***non-deterministic FA***, from state A the input “a” gives us **two choices**, either enter state B or state C |

**In case 1**, input **a** takes us from state A to only one next state B, this is an example of ***deterministic FA***. That is for each input there is only ONE next state to enter.

**In case 2**, at state A for input **a**, we have a choice to enter state B or state C (means there are more than one next state for input a ), this is an example of ***non-deterministic FA*** ( we are not determine whether to enter state B or state C)

When you design an FA to represent a grammar or a language, it is acceptable for the FA to be non-deterministic. But, when you want to write a program for that FA you have to make sure the FA is a deterministic. Following is a technique to convert non-deterministic FA to a deterministic FA.

**CONVERTING NON-DETERMINITIS FA TO A DETEMINITISTIC FA**

Example: Convert the given non-deterministic FA (NDFA) to a deterministic FA (DFA).

|  |  |
| --- | --- |
| **Non-deterministic FA: NDFA** | **Deterministic FA: DFA** |
| a b  a  B  A  This FA is non-deterministic. At state A  the input a issues two next options,  back to state A or enter state B. Hard to make the right decision when you write a program for this FA | Step 1. Construct the following table   |  |  |  | | --- | --- | --- | | state | Input a | Input b | | {A} | {A , B}  At state A with input a you have a choice to enter state A or state B. A🡪aA means from state A input a taks you back to A. A🡪aB, means from state A input a takes you to B. We write it as {A,B} | { }  At state A, input b is not declared | | {B} | { }  At state B, input a  is not declared | {B}  B🡪bB, means at state B input b takes you back to sate B | | A new state {A,B} is created, lets find the next state using inputs a and b at this state. Since {A,B}={A}U{B}, so the next state using input a is the same as the next state using a as input at {A} union with the next state using input a at {B} | | | | {A,B} | {A,B}={A}U{B}, input a  ={A,B}U{ }  ={A,B} | {A,B}={A}U{B}, input b  ={ }U{B}  ={B} | | There are no new states, the table is complete  New FA states are the first column of the table: {A},{B}, {A,B} | | |   Step 2. Now we use the table to construct a new FA which is the deterministic form of the original FA   1. The initial state remains the same: {A} 2. To decide which states are final states, go back to the original FA. Since the B was a final state in the original FA, the final states of this new FA are any state whose name contains B. Hence, {A,B} and {B} are final states of this DFA   a  a b    { A} {A,B} {B}  This machine has the same language as the original FA, and because it is deterministic, its easy to write a program to accepted or rejected any word. |

Note that while we convert NDFA to DFA, their grammar also changed from non-deterministic CFG (NDCFG) to a deterministic CFG (DCFG)

|  |  |
| --- | --- |
| **NDFA and its NDCFG** | **DFA and its DCFG** |
| a b  a  A B  NDCFG: A🡪aA |aB, B🡪bB | λ | a  a b    X={ A} Y={A,B} Z= {B}  For simplicity, rename the states  DCFG: X🡪aY, Y🡪aY |bZ |λ, B🡪λ |

**Example**. Convert the following NDFA to a b

a DFA. At the same time show how the a {B}

Grammar will change to a DCFG {A} a

1. initial state:{A} b a
2. Final state: {B} {C}

**Step 1.** Construct the transition table

|  |  |  |
| --- | --- | --- |
| **States** | **Input a** | **Input b** |
| {A} | {A,B} | { C } |
| {B} | { } | {B} |
| {C} | { C, B } | { } |
| New states: {A,B} and {C,B}, add them to the table | | |
| {A,B} | {A,B}={A}U{B} = {A,B}U{ } = {A,B} | {A,B}={A}U{B}={c}U{B}={C,B} |
| {C,B} | {C,B}={C}U{B}={C,B}U{ } = {C,B} | {C,B}={C}U{B}={ }U{B}={B} |
| No newer states. States of the new machine are {A}, {B}, {C}, {A,B}, {C,B} | | |

**Step 2**: Use the table to construct DFA

In the DFA, Initial state={A} and Final states are states whose B is a member of their name :{A,B}, {B}, and {C,B}

a

a {A,B}

{A} b a b

b

b a {C,B} {B}

{C}



To write its deterministic CFG, let X= {A}, Y={A,B}, Z={C,B}, D ={C}, and E={B}, then

X🡪aY | bD

X

/ \

a Y

/ \

b Z

/ \

a Z

/ \

b B

a b a b λ = w=abab

Y🡪aY | bZ | λ

D🡪aZ

Z🡪aZ | bB | λ

B🡪bB | λ

Use parsing tree to trace the word w=abab:



**CMPS 455 Names: Richard Gresham**

**Names: Sean McCarthy**

**Assignment No. 3** (70 points. Different forms of CFG, Regular and non-regular languages, regular and non-regular CFG, Deterministic and non-deterministic FAs, converting NDFA to DFA)

1. (10 points) Given the language L=(a + b)\*(ba\* + ab\*) a b

b a

1. Construct an FA for L (FA is given ) a b

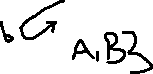
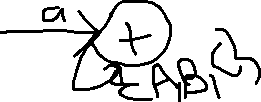
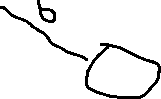
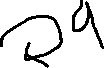
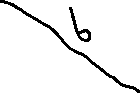
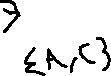
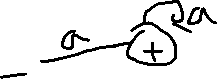
Diagram

Description automatically generated



1. Convert the non-deterministic FA (NDFA) to a deterministic FA (DFA)

|  |  |  |
| --- | --- | --- |
| States | a | b |
| {A} | {A,C} | {A,B} |
| {B} | {B} | {} |
| {C} | {} | {C} |
| {A,B} | {A U B} = {A,C} U {B} = {A,B,C} | {AUB} = {A,B} U {} = {A,B} |
| {A,C} | {A U C} = {A,C} | {A U C} = {A,B,C} |
| {A,B,C} | {AUBUC} = {A,B,C} | = {A,B,C} |



1. Use the DFA to write a deterministic CFG for L

{A} 🡪 a{A,C} | b{A,B}

{A,C} 🡺a{A,C} | b{A,B,C} | λ

{A,B} 🡺 a{A,B,C} | b{A,B} | λ



{A,B,C} 🡺 a{A,B,C} | b{A,B,C} | λ

1. (10 points) Given the following non-deterministic CFG :

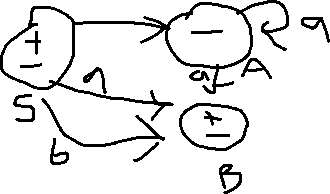


S 🡪 aA | aB | bB | λ

B🡪bB | λ

A🡪aA |aB

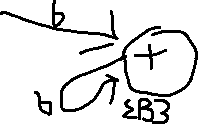
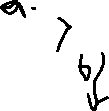
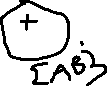
Convert the grammar to a deterministic CFG (hint Use the CFG to construct a non-deterministic FA, convert NDFA to DFA, and then write a new DCFG)



Step 2: transition table

|  |  |  |
| --- | --- | --- |
| States | a | b |
| {S} | {A,B} | {B} |
| {A} | {A,B} | { } |
| {B} | { } | {B} |
| {A.B} | {A,B} | {B} |

Step 3: DFA



Step 4: CFG of DFA

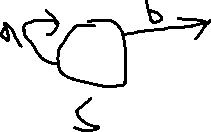
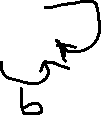
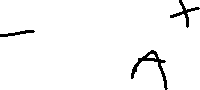
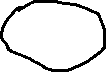
S 🡪 a{A,B} | b{B} | λ

{A,B} 🡺 a {A,B} | b {B} | λ

{B} 🡺 b{B} | λ

1. (10 points) Write a regular and non regular CFG for languages: ( i) L= a\*b (a+b)\* (ii) L=a\*b\*

(i)



Regular CFG:

S 🡪 aS | bA



A 🡺 aA | bA | λ

Non regular CFG:

S 🡪 {a} b{a + b}

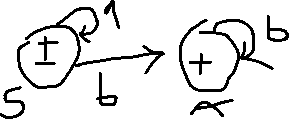


S 🡺 X b Y

X 🡺 aX | λ

Y 🡺 aY | bY | λ

(ii).



Regular CFG:

S 🡪 aS | bA | λ

A 🡺 bA | λ

Nonregular CFG:

S 🡪 {a} {b}



S 🡪 XY

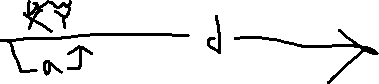
X 🡺 aX

X 🡺 λ

Y 🡺 bY

Y 🡺 λ

1. (20 points) Given the following EBNF grammars. ( i) draw their syntax diagram (ii)write each grammar in form of BNF
2. S🡪 [a] {b} d



S 🡪 X Y d

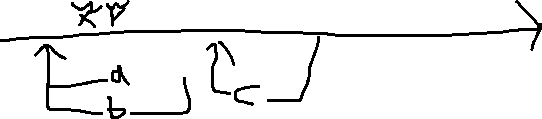
X 🡺 a

X 🡺 λ

Y 🡺 bY

Y 🡺 λ

1. S🡪 {a|b} {c}



S 🡪 XY

X 🡺 aX

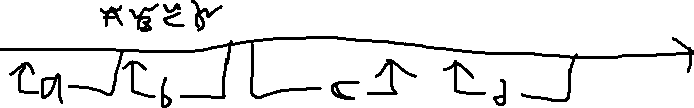
X 🡺 bX

X 🡺 λ

Y 🡺cY

Y 🡺 λ

1. S🡪 {a} {b} [c] {d}



S 🡪 ABCD

A 🡺 aA

A 🡺 λ

B 🡺 bB

B 🡺 λ

C 🡺 c

C 🡺 λ

D 🡺 dD

D 🡺 λ

**Programming (10 points each)**

1. Write a program to read one token at a

time from the given text file and determine whether the token is

1. A number
2. An identifier( must start with underscore or a letter, followed by

more letters, more digits , or more underscores

1. A reserved word, string reserved[5]={“while”, “for”, “switch”, “do”, “return” };

Sample output for #1

Token number identifier reserved word

K-mart no no no

23andMe no no no

456 yes no no

………..

456

Tax 2018

1. Given CFG: **S🡪aS |bB|cC** Write **a program** to determine whether an input string is accepted

**B🡪bB | aC | cD|λ** or rejected by the grammar.

**C🡪aS |bD |cD |λ**

**D🡪bD | aB| cC** Try input strings: w1=abbbcaaa$ , w2=ccccbbb$, w3=aabbcbbb$

**QUIZ in class**

K-mart

23andMe

456

Tax 2018

While

switch

do\_it

\_Fall\_20

\_Jan 19

# ------------------------------------------------------------------

#             Group names: Gresham, Richard and McCarthy, Sean

#             Assignment: No. 3

#             Due Date: February 16, 2022

#             Purpose: this program reads a file and determines whether a variable is a

#             ,identifier, and a reserved word.

# -----------------------------------------------------------------------

import os

def number\_check(value):

    # checks a value to see if it finds a float if not return false.

    try:

        float(value)

        return True

    except ValueError:

        return False

def tokenchecker(file, reserved):

    tokenlist = []

    Tabledict = {}

    dictcount = 0

    lines = (

        file.readlines()

    )  # python's built in line reader, reads an entire line at a time.

    count = 0

    for line in lines:

        count += 1

        line = line[

            :-1

        ]  # This deletes the extra space that is found by using the readlines() command at the end.

        tokenlist.append(line)  # add it to the token list.

        # print(f" line {count}: {line}") checks to confirm line read

    for token in tokenlist:

        # This loops throughout our token list and checks for each condition returing a table value based upong what the file had.

        tokeninfo = []

        numcheck = True

        identifiercheck = True

        reservedcheck = False

        if not number\_check(token):

            # checks to see if identifier passes number\_check.

            numcheck = False

        if not (

            (token[0] >= "a" and token[0] <= "z")

            or (token[0] >= "A" and token[0] <= "Z")

            or (token[0] == "\_")

        ):

            identifiercheck = False

        if identifiercheck == True:

            # checks to see if token passes identifier checks.

            for i in range(len(token)):

                if not (

                    (token[i] >= "a" and token[i] <= "z")

                    or (token[i] >= "A" and token[i] <= "Z")

                    or (token[i] == "\_")

                    or (token[i] >= "0" and token[i] <= "9")

                ):

                    identifiercheck = False

        for word in reserved:

            # checks to see if token passes reserved check.

            if token == word:

                reservedcheck = True

        tokeninfo.append(token)

        if numcheck == True:

            tokeninfo.append("yes")

        else:

            tokeninfo.append("no")

        if identifiercheck == True:

            tokeninfo.append("yes")

        else:

            tokeninfo.append("no")

        if reservedcheck == True:

            tokeninfo.append("yes")

        else:

            tokeninfo.append("no")

        Tabledict[dictcount] = tokeninfo

        dictcount = dictcount + 1

        # Below are commmands to make the table look nice.

    print(

        "{:<5} {:<8} {:<5} {:<8} {:<5}".format(

            "Pos", "Token", "number", "identifier", "reserved word"

        )

    )

    for k, v in Tabledict.items():

        Token, number, identifier, reservedword = v

        print(

            "{:<5} {:<8} {:<8} {:<8} {:<8}".format(

                k, Token, number, identifier, reservedword

            )

        )

def main():

    """This is my word checker program"""

    file = "WordCheck.txt"

    file = open(file, "r")

    # our list of reserved words.

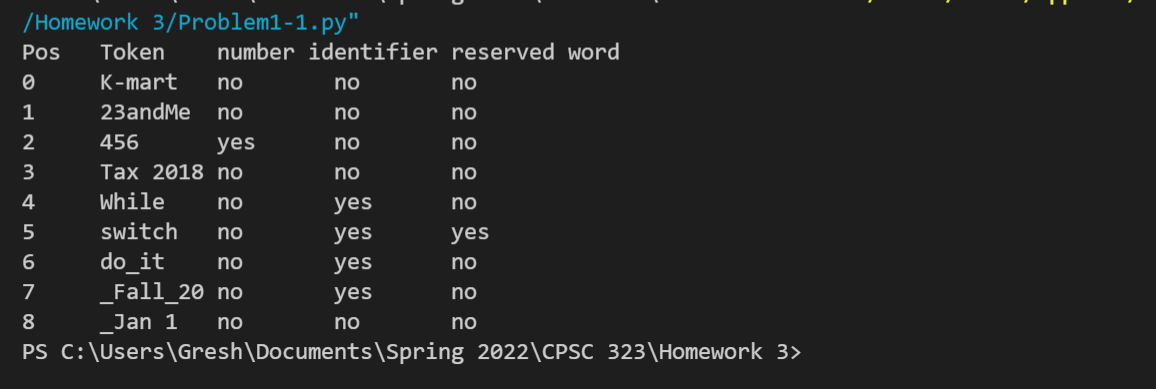
    reserved = ["while", "for", "switch", "do", "return"]

    tokenchecker(file, reserved)

    file.close()

if \_\_name\_\_ == "\_\_main\_\_":

    main()



.txt file of

K-mart

23andMe

456

Tax 2018

While

switch

do\_it

\_Fall\_20

\_Jan 19

# ------------------------------------------------------------------

#             Group names: Gresham, Richard and McCarthy, Sean

#             Assignment: No. 3

#             Due Date: February 16, 2022

#             Purpose: this program runs a CFG and determines whether

#             to accept or reject input string by grammar.

# -----------------------------------------------------------------------

from re import A

from sre\_parse import State

def String\_Checker(myinput):

    Initial\_State = 0

    transition\_table = [

        # a, b, c

        [0, 1, 2],  # S = 0

        [2, 1, 3],  # B = 1

        [0, 3, 3],  # C = 2

        [1, 3, 2],  # D = 3

    ]

    # The 2-D array above represents the FA and where the FA points to.

    CurrentState = Initial\_State

    #below checks each char input and changes the location of our current

    #state accordingly

    for char in myinput:

        if char == "a":

            CurrentState = transition\_table[CurrentState][0]

        elif char == "b":

            CurrentState = transition\_table[CurrentState][1]

        elif char == "c":

            CurrentState = transition\_table[CurrentState][2]

        elif char == "$":

            if CurrentState == 1 or CurrentState == 2:

                return True

            else:

                return False

        else:

            return False

def main():

    """This is my Grammar program"""

    word = input("Please input a Grammar to be validated: ")

    Final\_answer = String\_Checker(word)

    if Final\_answer == False:

        print("Rejected")

    else:

        print("Accepted")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

Text

Description automatically generated