Names: Angela Deleo
Roman Saddi
Jennah Kanan

Row: 4

CPSC 323-05
Final Project

1. Original Program

finalp1.txt ----------

```
program s2023;
//This program computes and prints the
value of a given expression //
 var
     // declare variables //
     p1 ,   p2q ,  pr  : integer ;
begin
     //initialize variables//
     p1     = 33 ;
     p2q =     412 ;
     pr=p1 + p2q;
     display ( pr ); // display pr

     //compute the value of
      the following expression //
     pr = p1 * ( p2q+ 2 * pr) ;
     display ( "value=",     pr ) ; // print the value of pr
  end.
```

After removing comments and ext

finalp2.txt ----------

```
program s2023;
var
p1 , p2q , pr : integer ;
begin
p1 = 33 ;
p2q = 412 ;
pr=p1 + p2q;
display ( pr );
pr = p1 * ( p2q+ 2 * pr) ;
display ( "value=", pr ) ;
end.
```

2. Original Grammar

<prog> → **program** <identifier>; **var** <dec-list> **begin** <stat-list> **end.**

<identifier> → <letter>{<letter>|<digit>}

<dec-list> → <dec> : <type> ;

<dec> → <identifier> , <dec> | <identifier>

<type> → **integer**

<stat-list> → <stat> | <stat> <stat-list>

<stat> → <write> | <assign>

<write> → **display** (“value=”, <identifier>) ; | **display** (<identifier>) ;

<assign> → <identifier> = <expr> ;

<expr> → <expr> + <term> | <expr> - <term> | <term>

<term> → <term> * <factor> | <term> / <factor> | <factor>

<factor> → (<expr>)

<factor> → <identifier> | <number>

<number> → <sign> <digit>{<digit>}

<sign> → + | - | λ

<digit> → 0 | 1 | 2 | … | 9

<letter> → p | q | r | s

- *Non-terminals* = { <prog> <identifier> <dec-list> <dec> <type> <stat-list> <stat> <write> <assign> <expr> <term> <factor> <number> <sign> <digit> <letter> }

- *Terminals* = { ; : , + - * / = 0 1 2 … 9 p q r s “value=”, ( ) λ }

- *Reserved words* = { **program var begin display end.** }
  * note Reserved Words are considered terminals. *

3. Original Grammar in BNF Form

<prog> → **program** <identifier>; **var** <dec-list> **begin** <stat-list> **end.**

<identifier> → <letter> <id>

<id> → <letter> <id> | <digit> <id> | λ

<dec-list> → <dec> : <type> ;

<dec> → <identifier> , <dec> | <identifier>

<type> → **integer**

<stat-list> → <stat> | <stat> <stat-list>

<stat> → <write> | <assign>

<write> → **display** (“value=”, <identifier>) ; | **display** (<identifier>) ;

<assign> → <identifier> = <expr> ;

<expr> → <expr> + <term> | <expr> - <term> | <term>

<term> → <term> * <factor> | <term> / <factor> | <factor>

<factor> → (<expr>)

<factor> → <identifier> | <number>

<number> → <sign> <num> | <num>

<num> → <digit> <num> | λ

<sign> → + | - | λ

<digit> → 0 | 1 | 2 | … | 9

<letter> → p | q | r | s

4. Preparation for Predictive Parsing Table

Final Form of Grammar in BNF Form (after *removing left-recursion*):

<prog> → **program** <identifier>; **var** <dec-list> **begin** <stat-list> **end.**
<identifier> → <letter> <id>
<id> → <letter> <id> | <digit> <id> | λ
<dec-list> → <dec> : <type> ;
<dec> → <identifier> <dec2>
<dec2> → , <identifier> | λ
<type> → **integer**
<stat-list> → <stat> <stat-list2>
<stat-list2> → <stat> <stat-list2> | λ
<stat> → <write> <stat> | <assign> <stat>
<write> → **display** (<write-list>) ;
<write-list> → “value=”, <identifier> | <identifier>
<assign> → <identifier> = <expr> ;
<expr> → <term> <Q>
<Q> → + <term> <Q> | - <term> <Q> | λ
<term> → <factor> <R>
<R> → * <factor> <R> | / <factor> <R> | λ
<factor> → (<expr>) | <identifier> | <number>
<number> → <sign> <num> | <num>
<num> → <digit> <num> | λ
<sign> → + | - | λ
<digit> → 0 | 1 | 2 | … | 9
<letter> → p | q | r | s

5. Members of FIRST and FOLLOW

State	New Name	FIRST	FOLLOW
<prog>	P	{ program }	{ $ }
<identifier>	I	{ p q r s }	{) ; : , = p q r s 0 … 9 }
<id>	J	{ p q r s 0 … 9 }	{) ; : , = + - * / }
<dec-list>	H	{ var p q r s }	{ begin }
<dec>	C	{ p q r s }	{ : }
<dec2>	K	{ , }	{ : }
<type>	Y	{ integer }	{ ; }
<stat-list>	G	{ display p q r s }	{ end. }
<stat-list2>	O	{ display p q r s }	{ end. display p q r s }
<stat>	S	{ display p q r s }	{ end. display p q r s }
<write>	W	{ display }	{ ; end.}
<write-list>	B	{ “value=”, p q r s }	{) }
<assign>	A	{ p q r s }	{ ; end. }
<expr>	E	{ (p q r s + - 0 … 9 }	{) ; }
<Q>	Q	{ + - }	{) ; }
<term>	T	{ (p q r s + - * / 0 … 9 }	{) ; + - }
<R>	R	{ * / }	{) ; + - }
<factor>	F	{ (p q r s + - 0 … 9 }	{) ; + - * / }
<number>	N	{ + - 0 … 9 }	{) ; + - * / }
<num>	M	{ 0 … 9 }	{) ; + - * / }
<sign>	U	{ + - }	{ 0 … 9 }
<digit>	D	{ 0 … 9 }	{) ; , = p q r s 0 … 9 }
<letter>	L	{ p q r s }	{) ; , = p q r s 0 … 9 }

6. Predictive Parsing Table

Terminals (not including reserved words):

	;	:	,	(	)	“value=”,	=	+	-	*	/	0	…	9	p	q	r	s
P																		
I															LJ	LJ	LJ	LJ
J	λ	λ	λ		λ		λ	λ	λ	λ	λ	DJ	DJ	DJ	LJ	LJ	LJ	LJ
H															C : Y;	C : Y;	C : Y;	C : Y;
C															IK	IK	IK	IK
K		λ	,K												IK	IK	IK	IK
Y																		
G															SO	SO	SO	SO
O															λ	λ	λ	λ
S															AS	AS	AS	AS
W																		
B						“value=” , I									I	I	I	I
A															I = E;	I = E;	I = E;	I = E;
E				T Q				+TQ	-TQ			TQ	TQ	TQ	TQ	TQ	TQ	TQ
Q	λ				λ			+TQ	-TQ									
T				F R						*FR	/FR	FR	FR	FR	FR	FR	FR	FR
R	λ				λ			λ	λ	*FR	/FR							
F				(E)				N	N			N	N	N	I	I	I	I
N								UM	UM			M	M	M				
M	λ				λ			λ	λ	λ	λ	DM	DM	DM				
U								+	-			λ	λ	λ				
D												0	…	9				
L															p	q	r	S

Terminals, Only Reserved Words:

	program	**var**	**begin**	**end.**	**integer**	**display**
P	program I ; var H begin G end.					
I						
J						
H						
C						
K						
Y					integer	
G						SO
O				λ		λ
S						WS
W						display (B) ;
B						
A						
E						
Q						
T						
R						
F						
N						
M						
U						
D						
L						

7. Complete Program

main.py

```
from settingUp import *
set_up_file()

from grammar import *
from create_table import *
from convertPython import *
from outputPython import *

# initialize file to be read and parsing table to be used
text = txt_tostr('finalp2.txt')
parsing_table = initialize_parsing_table()

# check grammar of text in file
result = check_grammar(parsing_table, text)

if result:
   print("\nAccept\n")  # grammar was valid
   print("\nAfter conversion to Python3: \n")

   # call code to translate to python here
   toPython()

   # call converted code here
   eval(pull_function() + "()")

else:
   print("\nNot Accept")  # grammar was invalid
```

settingUp.py

```
import nltk

def set_up_file():
    """ Set up file for parsing. Remove blanks and comments."""
    # open the input file (finalpt1.txt)
    with open("finalp1.txt", "r") as file:
        # read content of the file
        content = file.readlines()

    # remove all comments, blank lines, and clean up spaces
    new_content = []
    prevLine=""
    addedLine=""
    addLineFlag = True

    #iterate through the string
    for line in content:
        addedLine=line
        addLineFlag = True

        # remove all the comments in the form; //comment, //comment//, and block comments. *can't have aspace
after*
        if addedLine.endswith("//\n") and prevLine.startswith("//"):
                addLineFlag = False
                continue
        if addedLine.endswith("// ") and prevLine.startswith("//"):
                addLineFlag = False
                continue
        if "//" in addedLine and "//" in prevLine:
                addLineFlag = True
                continue
        elif "//" in addedLine:
        addedLine = addedLine[:addedLine.index("//")]

        # remove all the blank lines
        if addedLine.strip() == "":
                prevLine = line
                addLineFlag = False
                continue

        # clean up all the spaces
        addedLine = " ".join(addedLine.split())

        # append the modified line to the new content
        if addLineFlag is True:
                new_content.append(addedLine)

        prevLine=line

    # open the output file
```

```
    with open("finalp2.txt", "w") as file:
        # write modified content to the output file
        file.write("\n".join(new_content))


def txt_tostr(file_name):
    """Convert text in a file to a long string"""

    input_text = ""
    with open(file_name, 'r') as f:
     input_text = f.read()

    return input_text


def init_text(text, inputted_rw, reserved_words, digits, letters):
    """Initialize text for parsing purposes."""

    input_text = []

    # split input text into separate characters
    for word in text.split():
     if word in inputted_rw or word in reserved_words:
        input_text.append(word)
     elif word == '"value=",':
        input_text.append('"value=",')
     else:
        for char in word:
                input_text.append(char)

    return input_text


def get_inputted_reserved_words(text, letters, variable_names):
    """Get inputted reserved words to check if misspelt"""

    # get inputted reserved words
    inputted_rw_temp = text.translate({ord(c): " " for c in '"*();:,/-=+ '})
    inputted_rw = []

    # get all inputted reserved words
    for rw in inputted_rw_temp.split():
     if rw in letters:
        continue
     elif rw == '"value=",':
        continue
     elif rw in variable_names:
        continue
     else:
        inputted_rw.append(rw)

    # remove any digits
```

```
    inputted_rw = [
     x for x in inputted_rw
     if not (x.isdigit() or x[0] == '-' and x[1:].isdigit())
    ]

    # remove any duplicates
    in_rw_no_dups = []
    [
     in_rw_no_dups.append(item) for item in inputted_rw
     if item not in in_rw_no_dups
    ]

    return in_rw_no_dups


def get_variable_names(text):
    """Get names of variables from the text after reserved word var and before reserved word begin. Include
variable name of the program. """

    variable_names = []

    # split up input text including variable names
    for word in text.split():
     if word == 'program':
        k = text.split().index(word)

        for j in range(k + 1, len(text.split())):

                if nltk.edit_distance("var", text.split()[j]) <= 2:
                 break
                else:
                 if ';' in text.split()[j]:
                        variable_names.append(text.split()[j].replace(';', ''))
                 else:
                        variable_names.append(text.split()[j])

     elif nltk.edit_distance("var", word) > 2:
        continue

     else:
        special_chars = [',', ':', ';', '=', '+', '-', '*', '/', '(', ')']

        k = text.split().index(word)

        for j in range(k + 1, len(text.split())):

                if text.split()[j] == 'integer' or text.split()[j].startswith('i'):
                 break
                elif text.split()[j] not in ', :':
                 variable_names.append(text.split()[j])

    return variable_names
```

```
def get_undefined_vars(text, variable_names, inputted_rw, reserved_words,
                                                    digits, letters):
    """Get any undefined variable for error checking"""

    # get defined variables
    defined_variables = []

    while "" in variable_names:
     variable_names.remove("")

    for varname in variable_names:
     defined_variables.append(
         varname.translate({ord(c): ""
                                                     for c in "*();:,/-=+ "}))

    while "" in defined_variables:
     defined_variables.remove("")

    defined_variables = set(defined_variables)

    # get undefined variables
    # parse thru everything after begin
    undefined_variables_temp = []
    undefined_variables = []

    # check all used variables after reserved word begin to see if any undefined variables are there
    for word in text.split():
     if word != 'begin':
         continue

     else:
         special_chars = [
                '"', 'value', ',', ':', ';', '=', '+', '-', '*', '/', '(', ')'
         ]

         k = text.split().index(word)
         for j in range(k + 1, len(text.split())):

                if text.split()[j] == 'end.':
                 break
                elif text.split()[j] == 'display':
                 continue
                elif text.split()[j] == '"value=",':
                 continue
                elif text.split()[j] in special_chars:
                 continue
                elif text.split()[j] in digits:
                 continue
                else:
                 for t in text.split()[j]:
```

```
                    if t in digits:
                        continue
                undefined_variables_temp.append(text.split()[j])

for undef_var in undefined_variables_temp:
 undefined_variables.append(
     undef_var.translate({ord(c): " "
                                                    for c in '"*();:,/-=+'}))

while 'value' in undefined_variables:
 undefined_variables.remove('value')

# make sure the undefined vars dont include reserved words
for undef_var in undefined_variables:
 if undef_var in inputted_rw or undef_var in reserved_words:
     undefined_variables.remove(undef_var)

undefined_variables = ' '.join(undefined_variables)
undefined_variables = undefined_variables.split()

# remove defined variables from undefined
for def_var in defined_variables:
 while def_var in undefined_variables:
     undefined_variables.remove(def_var)

undefined_variables = [
 x for x in undefined_variables
 if not (x.isdigit() or x[0] == '-' and x[1:].isdigit())
]

return undefined_variables
```

create_table.py

```
from tabulate import tabulate


def initialize_parsing_table():
    """ Initialize already calculated parsing table """

    parsing_table = [[' ' for x in range(31)] for y in range(23)]

    # state P := <prog>
    # row P, col program
    parsing_table[0][25] = 'program I ; var H begin G end.'

    # state I := <identifier>
    # row I, cols p, q, r, s
    for i in range(21, 25):
        parsing_table[1][i] = 'L J'

    # state J from removing left-recursion
    # row J, cols ;, :, ,
    for i in range(0, 3):
        parsing_table[2][i] = 'lamb'
    # row J, col )
    parsing_table[2][4] = 'lamb'
    # row J, col =, +, -, *, /
    for i in range(6, 11):
        parsing_table[2][i] = 'lamb'
    # row J, cols 0-9
    for i in range(11, 21):
        parsing_table[2][i] = 'D J'
    # row J, cols p, q, r, s
    for i in range(21, 25):
        parsing_table[2][i] = 'L J'

    # state H := <dec-list>
    # row H, cols p, q, r, s
    for i in range(21, 25):
        parsing_table[3][i] = 'C : Y ;'

    # state C := <dec>
    # row C, cols p, q, r, s
    for i in range(21, 25):
        parsing_table[4][i] = 'I K'

    # state K from removing left-recursion
    # row K, col :
    parsing_table[5][1] = 'lamb'
    # row K, col ,
    parsing_table[5][2] = ', K'
    # row K, cols p, q, r, s
    for i in range(21, 25):
```

```
    parsing_table[5][i] = 'I K'

# state Y := <type>
# row Y, col integer
parsing_table[6][29] = 'integer'

# state G := <stat-list>
# row G, cols p, q, r, s
for i in range(21, 25):
    parsing_table[7][i] = 'S O'
# row G, col display
parsing_table[7][30] = 'S O'

# state O from removing left-recursion
# row O, cols p, q, r, s
for i in range(21, 25):
    parsing_table[8][i] = 'lamb'
# row O, col end.
parsing_table[8][28] = 'lamb'
# row O, col display
parsing_table[8][30] = 'lamb'

# state S := <stat>
# row S, col p, q, r, s
for i in range(21, 25):
    parsing_table[9][i] = 'A S'
# row S, col display
parsing_table[9][30] = 'W S'

# state W := <write>
# row W, col display
parsing_table[10][30] = 'display ( B ) ;'

# state B from removing left-recursion
# row B, col "value="
parsing_table[11][5] = '"value=" , I'
# row B, col p, q, r, s
for i in range(21, 25):
    parsing_table[11][i] = 'I'

# state A := <assign>
# row A, col p, q, r, s
for i in range(21, 25):
    parsing_table[12][i] = 'I = E ;'

# state E := <expr>
# row E, col )
parsing_table[13][3] = 'T Q'
# row E, col +
parsing_table[13][7] = '+ T Q'
# row E, col -
parsing_table[13][8] = '- T Q'
```

```
# row E, col 0-9, p, q, r, s
for i in range(11, 25):
    parsing_table[13][i] = 'T Q'

# state Q from removing left-recursion
# row Q, col ;
parsing_table[14][0] = 'lamb'
# row Q, col )
parsing_table[14][4] = 'lamb'
# row Q, col +
parsing_table[14][7] = '+ T Q'
# row Q, col -
parsing_table[14][8] = '- T Q'

# state T := <term>
# row T, col (
parsing_table[15][3] = 'F R'
# row T, col *
parsing_table[15][9] = '* F R'
# row T, col /
parsing_table[15][10] = '/ F R'
# row T, col 0-9, p, q, r, s
for i in range(11, 25):
    parsing_table[15][i] = 'F R'

# state R from removing left-recursion
# row R, col ;
parsing_table[16][0] = 'lamb'
# row R, col )
parsing_table[16][4] = 'lamb'
# row R, col +
parsing_table[16][7] = 'lamb'
# row R, col -
parsing_table[16][8] = 'lamb'
# row R, col *
parsing_table[16][9] = '* F R'
# row R, col /
parsing_table[16][10] = '/ F R'

# state F := <factor>
# row F, col (
parsing_table[17][3] = '( E )'
# row F, col +
parsing_table[17][7] = 'N'
# row F, col -
parsing_table[17][8] = 'N'
# row F, cols 0-9
for i in range(11, 21):
    parsing_table[17][i] = 'N'
# row F, cols p, q, r, s
for i in range(21, 25):
    parsing_table[17][i] = 'I'
```

```
# state N := <number>
# row N, cols +, -
for i in range(7, 9):
    parsing_table[18][i] = 'U M'
# row N, cols 0-9
for i in range(11, 21):
    parsing_table[18][i] = 'M'

# state M from removing left-recursion
# row M, col ;
parsing_table[19][0] = 'lamb'
# row M, col )
parsing_table[19][4] = 'lamb'
# row M, cols +, -, *, /
for i in range(7, 11):
    parsing_table[19][i] = 'lamb'
# row M, cols 0-9
for i in range(11, 21):
    parsing_table[19][i] = 'D M'

# state U := <sign>
# row U, col +
parsing_table[20][7] = '+'
# row U, col -
parsing_table[20][8] = '-'
# row U, cols 0-9
for i in range(11, 21):
    parsing_table[20][i] = 'lamb'

# state D := <digit>
# row D, cols 0-9
for i in range(11, 21):
    digit = i - 11
    parsing_table[21][i] = digit

# state L := <letter>
# row L, col p
parsing_table[22][21] = 'p'
# row L, col q
parsing_table[22][22] = 'q'
# row L, col p
parsing_table[22][23] = 'r'
# row L, col p
parsing_table[22][24] = 's'

# saving a nice looking version of the table

nonterminals = [
    'P', 'I', 'J', 'H', 'C', 'K', 'Y', 'G', 'O', 'S', 'W', 'B', 'A', 'E', 'Q',
    'T', 'R', 'F', 'N', 'M', 'U', 'D', 'L'
]
```

```
pretty_table = tabulate(parsing_table,
                                        headers=[
                                               ';', ':', ',', '(', ')', '"value="', '=', '+', '-',
                                               '*', '/', '0', '1', '2', '3', '4', '5', '6', '7',
                                               '8', '9', 'p', 'q', 'r', 's', 'program', 'var',
                                               'begin', 'end.', 'integer', 'display'
                                                ],
                                       showindex=nonterminals,
                                       tablefmt="grid")
# save to txt
with open('parsing_table.txt', 'w') as f:
    f.write(pretty_table)

# return parsing_table so we can use it to check grammar
return parsing_table
```

grammar.py

```
import errors
from settingUp import *
from create_table import *
from check_arithmetic import *


def check_grammar(parsing_table, text):
    """ Check grammar of a given text using a given parsing table """

    # initialize stack, counter for parsing, reserved_words, digits, and letters
    stack = []
    i = 0
    reserved_words = [
        'program', 'var', 'begin', 'end.', 'display', 'integer', '"value=",'
    ]
    digits = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
    letters = ['p', 'q', 'r', 's']

    # prepare text for parsing, call all settingUp functions except for file based ones
    # get variable names, inputted reserved words, undefined variables, as well as the input text
    variable_names = get_variable_names(text)
    inputted_rw = get_inputted_reserved_words(text, letters, variable_names)
    undefined_variables = get_undefined_vars(text, variable_names, inputted_rw,

                                        reserved_words, digits, letters)
    input_text = init_text(text, inputted_rw, reserved_words, digits, letters)

    # if there are undefined variables in the text, raise an error
    if undefined_variables:
        raise errors.MissingArg('', 0, '', undefined_variables)

    # begin tracing
    stack.append('P')  # push P

    print(f"current stack: {stack}")
    current = stack.pop()  # pop P

    # parse thru entire input_text
    while i < len(input_text):
        print(f"\nreading {input_text[i]}")
        print(f"just popped {current}")

        # state P
        if current == 'P':
               # case: [row P, col program I val H begin G]
               if input_text[i] == reserved_words[0]:
                      for j in range(len(parsing_table[0][25].split()) - 1, -1, -1):
                             stack.append(parsing_table[0][25].split()[j])

                      print(f"current stack: {stack}")
```

```
                current = stack.pop()  # pop program

        else:
                raise errors.MissingArg('program', i, inputted_rw[0])

# state I
elif current == 'I':
        # case: [row I, col p, q, r, s]
        if input_text[i] in letters:
                stack.append(parsing_table[1][21].split()[1])  # push J
                stack.append(parsing_table[1][21].split()[0])  # push L

                print(f"current stack: {stack}")
                current = stack.pop()  # pop L
        else:
                # identifier started with a number
                if input_text[i].isdigit():
                        print("The start of an identifier cannot be numerical.")
                elif input_text[i] in '[@_!#$%^&*()<>?/\|}{~:]':
                        print(
                                "The start of an identifier cannot contain special characters.")
                elif input_text[i] == ';':
                        raise errors.MissingArg('', i)
                return False

# state J
elif current == 'J':
        # case: [row J, col p, q, r, s]
        if input_text[i] in letters:
                stack.append(parsing_table[2][21].split()[1])  # push J
                stack.append(parsing_table[2][21].split()[0])  # push L

                print(f"current stack: {stack}")
                current = stack.pop()  # pop L

        # case: [row J, col ; : , ) = * /]
        elif input_text[i] in '; : , ) = + - * /':
                stack.append(parsing_table[2][0].split()[0])  # push lambda

                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda

        # case: [row J, col 0-9]
        elif input_text[i] in digits:
                stack.append(parsing_table[2][11].split()[1])  # push J
                stack.append(parsing_table[2][11].split()[0])  # push D

                print(f"current stack: {stack}")
                current = stack.pop()  # pop D

        else:
                if input_text[i] not in letters or list(
```

```
                        input_text[i])[0] not in digits:
                    if input_text[i] == 'var':
                        raise errors.MissingArg(';', i)
                    elif input_text[i] == 'integer':
                        raise errors.MissingArg(':', i)
                    elif list(input_text[i - 1])[0] in letters or list(
                            input_text[i - 1])[0] in digits:
                        if input_text[i] not in '+ -':
                            raise errors.MissingArg('=', i)
            return False

    # state H
    elif current == 'H':
        # case: [row H, col p, q, r, s]
        if input_text[i] in letters:
            stack.append(parsing_table[3][21].split()[3])  # push ;
            stack.append(parsing_table[3][21].split()[2])  # push Y
            stack.append(parsing_table[3][21].split()[1])  # push :
            stack.append(parsing_table[3][21].split()[0])  # push C

            print(f"current stack: {stack}")
            current = stack.pop()  # pop C

        else:
            if input_text[i] == 'integer':
                raise errors.MissingArg(':', i)
            else:
                print("Variable names must begin with an alphabetical character.")
            return False

    # state C
    elif current == 'C':
        # case: [row C, col p, q, r, s]
        if input_text[i] in letters:
            stack.append(parsing_table[4][21].split()[1])  # push K
            stack.append(parsing_table[4][21].split()[0])  # push I

            print(f"current stack: {stack}")
            current = stack.pop()  # pop I

        else:
            if input_text[i] == ' ':
                raise errors.MissingArg('', i)
            else:
                print("Variable names must begin with an alphabetical character.")
            return False

    # state K
    elif current == 'K':
        # case: [row K, col :]
        if input_text[i] == ':':
            stack.append(parsing_table[5][1].split()[0])  # push lambda
```

```
                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda

        elif input_text[i] == ',':
                stack.append(parsing_table[5][2].split()[1])  # push K
                stack.append(parsing_table[5][2].split()[0])  # push ,

                print(f"current stack: {stack}")
                current = stack.pop()  # pop ,

        # case: [row K, col p, q, r, s]
        elif input_text[i] in letters:
                stack.append(parsing_table[5][21].split()[1])  # push K
                stack.append(parsing_table[5][21].split()[0])  # push I

                print(f"current stack: {stack}")
                current = stack.pop()  # pop I

        else:
                raise errors.MissingArg(',', i)

# state Y
elif current == 'Y':
        # case: [row Y, col integer]
        if input_text[i] == 'integer':
                stack.append(parsing_table[6][29].split()[0])  # push integer

                print(f"current stack: {stack}")
                current = stack.pop()  # pop integer

        else:
                raise errors.MissingArg('integer', i, inputted_rw[2])

# state G
elif current == 'G':
        # case: [row G, col p, q, r, s]
        if input_text[i] in letters:
                stack.append(parsing_table[7][21].split()[1])  # push O
                stack.append(parsing_table[7][21].split()[0])  # push S

                print(f"current stack: {stack}")
                current = stack.pop()  # pop S

        elif input_text[i] == 'display':
                stack.append(parsing_table[7][30].split()[0])  # push O
                stack.append(parsing_table[7][30].split()[0])  # push S

                print(f"current stack: {stack}")
                current = stack.pop()  # pop S

        else:
```

```
                if input_text[i] not in letters:
                        raise errors.MissingArg('display', i, inputted_rw[4])
                return False

# state O
elif current == 'O':
        # case: [row O, col p, q, r, s]
        if input_text[i] in letters:
                stack.append(parsing_table[8][21].split()[0])  # push lambda

                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda

        elif input_text[i] == 'display':
                stack.append(parsing_table[8][30].split()[0])  # push lambda

                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda

        elif input_text[i] == 'end.':
                stack.append(parsing_table[8][28].split()[0])  # push lambda

                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda

        else:

                if input_text[i] not in letters:
                        if input_text[i] != 'display':
                                raise errors.MissingArg('end.', i, inputted_rw[5])
                        else:
                                raise errors.MissingArg('display', i, inputted_rw[4])

# state S
elif current == 'S':
        # case: [row S, col p, q, r, s]
        if input_text[i] in letters:
                stack.append(parsing_table[9][21].split()[1])  # push S
                stack.append(parsing_table[9][21].split()[0])  # push A

                print(f"current stack: {stack}")
                current = stack.pop()  # pop A

        elif input_text[i] == 'display':
                stack.append(parsing_table[9][30].split()[1])  # push S
                stack.append(parsing_table[9][30].split()[0])  # push W

                print(f"current stack: {stack}")
                current = stack.pop()  # pop display

        elif input_text[i] == 'end.':
                stack.append(parsing_table[8][28].split()[0])  # push lambda
```

```
                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda

        else:
                if input_text[i] == 'end':
                        raise errors.MissingArg('.', 'the end')
                elif input_text[i].startswith('e'):
                        raise errors.MissingArg('end.', i, inputted_rw[6])
                elif input_text[i].startswith('d'):
                        raise errors.MissingArg('display', i, inputted_rw[4])
                elif input_text[i] == '(':
                        raise errors.MissingArg('display', i, inputted_rw[4])
                return False

# state W
elif current == 'W':
        # case: [row W, col display]
        if input_text[i] == 'display':
                stack.append(parsing_table[10][30].split()[4])  # push ;
                stack.append(parsing_table[10][30].split()[3])  # push )
                stack.append(parsing_table[10][30].split()[2])  # push B
                stack.append(parsing_table[10][30].split()[1])  # push (
                stack.append(parsing_table[10][30].split()[0])  # push display

                print(f"current stack: {stack}")
                current = stack.pop()  # pop display
        else:
                raise errors.MissingArg('display', i, inputted_rw[4])

# state B
elif current == 'B':
        # case: [row B, col "value="]
        if input_text[i] == '"value=",':
                stack.append(parsing_table[11][5].split()[2])  # push I
                stack.append(''.join(
                        parsing_table[11][5].split()[0:2]))  # push "value=",

                print(f"current stack: {stack}")
                current = stack.pop()  # pop "value=",

        # case: [row B, col p q r s]
        elif input_text[i] in letters:
                stack.append(parsing_table[11][21].split()[0])  # push I

                print(f"current stack: {stack}")
                current = stack.pop()  # pop I

        else:
                return False

# state A
```

```
elif current == 'A':
	# case: [row A, col p q r s]
	if input_text[i] in letters:
		stack.append(parsing_table[12][21].split()[3])  # push ;
		stack.append(parsing_table[12][21].split()[2])  # push E
		stack.append(parsing_table[12][21].split()[1])  # push =
		stack.append(parsing_table[12][21].split()[0])  # push I

		print(f"current stack: {stack}")
		current = stack.pop()  # pop I

	else:
		raise errors.MissingArg('', i)

# state E
elif current == 'E':
	# case: [row E, col p, q, r, s]
	if input_text[i] in letters:
		stack.append(parsing_table[13][21].split()[1])  # push Q
		stack.append(parsing_table[13][21].split()[0])  # push T

		print(f"current stack: {stack}")
		current = stack.pop()  # pop T

	# case: [row E, col + - (]
	elif input_text[i] in '+ - (':
		print(parsing_table[13][3].split())
		stack.append(parsing_table[13][3].split()[1])  # push Q
		stack.append(parsing_table[13][3].split()[0])  # push T

		print(f"current stack: {stack}")
		current = stack.pop()  # pop T

	# case: [row E, col 0-9]
	elif input_text[i] in digits:
		stack.append(parsing_table[13][11].split()[1])  # push Q
		stack.append(parsing_table[13][11].split()[0])  # push T

		print(f"current stack: {stack}")
		current = stack.pop()  # pop T

	else:
		raise errors.MissingArg('(', i)

# state Q
elif current == 'Q':
	# case: [row Q, col ;]
	if input_text[i] == ';':
		stack.append(parsing_table[14][0].split()[0])  # push lambda

		print(f"current stack: {stack}")
		current = stack.pop()  # pop lambda
```

```
        # case: [row Q, col )]
        elif input_text[i] == ')':
                stack.append(parsing_table[14][4].split()[0])  # push lambda

                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda

        # case: [row Q, col +]
        elif input_text[i] == '+':
                stack.append(parsing_table[14][7].split()[2])  # push Q
                stack.append(parsing_table[14][7].split()[1])  # push T
                stack.append(parsing_table[14][7].split()[0])  # push +

                print(f"current stack: {stack}")
                current = stack.pop()  # pop +

        # case: [row Q, col -]
        elif input_text[i] == '-':
                stack.append(parsing_table[14][8].split()[2])  # push Q
                stack.append(parsing_table[14][8].split()[1])  # push T
                stack.append(parsing_table[14][8].split()[0])  # push -

                print(f"current stack: {stack}")
                current = stack.pop()  # pop -

        else:
                return False

# state T
elif current == 'T':
        # case: [row T, col p, q, r, s]
        if input_text[i] in letters:
                stack.append(parsing_table[15][21].split()[1])  # push R
                stack.append(parsing_table[15][21].split()[0])  # push F

                print(f"current stack: {stack}")
                current = stack.pop()  # pop F

        # case: [row T, col + - (]
        elif input_text[i] in '+ - (':
                stack.append(parsing_table[15][3].split()[1])  # push R
                stack.append(parsing_table[15][3].split()[0])  # push F

                print(f"current stack: {stack}")
                current = stack.pop()  # pop F

        # case: [row T, col 0-9]
        elif input_text[i] in digits:
                stack.append(parsing_table[15][11].split()[1])  # push R
                stack.append(parsing_table[15][11].split()[0])  # push F
```

```
                print(f"current stack: {stack}")
                current = stack.pop()  # pop F

        else:
                return False

# state R
elif current == 'R':
        # case: [row R, col ;]
        if input_text[i] == ';':
                stack.append(parsing_table[16][0].split()[0])  # push lambda

                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda

        # case: [row R, col )]
        elif input_text[i] == ')':
                stack.append(parsing_table[16][4].split()[0])  # push lambda

                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda

        # case: [row R, col +]
        elif input_text[i] == '+':
                stack.append(parsing_table[16][7].split()[0])  # push lambda

                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda

        # case: [row R, col -]
        elif input_text[i] == '-':
                stack.append(parsing_table[16][8].split()[0])  # push lambda

                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda

        # case: [row R, col *]
        elif input_text[i] == '*':
                stack.append(parsing_table[16][9].split()[2])  # push R
                stack.append(parsing_table[16][9].split()[1])  # push F
                stack.append(parsing_table[16][9].split()[0])  # push *

                print(f"current stack: {stack}")
                current = stack.pop()  # pop *

        # case: [row R, col /]
        elif input_text[i] == '/':
                stack.append(parsing_table[16][10].split()[2])  # push R
                stack.append(parsing_table[16][10].split()[1])  # push F
                stack.append(parsing_table[16][10].split()[0])  # push /

                print(f"current stack: {stack}")
```

```
                current = stack.pop()  # pop /

        else:
                return False

# state F
elif current == 'F':
        # case: [row F, col (]
        if input_text[i] == '(':
                stack.append(parsing_table[17][3].split()[2])  # push )
                stack.append(parsing_table[17][3].split()[1])  # push E
                stack.append(parsing_table[17][3].split()[0])  # push (

                print(f"current stack: {stack}")
                current = stack.pop()  # pop (

        # case: [row F, col + -]
        elif input_text[i] in '+ -':
                stack.append(parsing_table[17][7].split()[0])  # push N

                print(f"current stack: {stack}")
                current = stack.pop()  # pop N

        # case: [row F, col 0-9]
        elif input_text[i] in digits:
                stack.append(parsing_table[17][11].split()[0])  # push N

                print(f"current stack: {stack}")
                current = stack.pop()  # pop N

        # case: [row F, col p q r s]
        elif input_text[i] in letters:
                stack.append(parsing_table[17][21].split()[0])  # push I

                print(f"current stack: {stack}")
                current = stack.pop()  # pop I

        else:
                raise errors.MissingArg('(', i)

# state N
elif current == 'N':
        # case: [row N, col + -]
        if input_text[i] in '+ -':
                stack.append(parsing_table[18][7].split()[1])  # push M
                stack.append(parsing_table[18][7].split()[0])  # push U

                print(f"current stack: {stack}")
                current = stack.pop()  # pop U

        # case: [row N, col 0-9]
        elif input_text[i] in digits:
```

```
                stack.append(parsing_table[18][11].split()[0])  # push M

                print(f"current stack: {stack}")
                current = stack.pop()  # pop M

        else:
                return False

# state M
elif current == 'M':
        # case: [row M, col ;]
        if input_text[i] == ';':
                stack.append(parsing_table[19][0].split()[0])  # push lambda

                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda

        # case: [row M, col )]
        elif input_text[i] == ')':
                stack.append(parsing_table[19][4].split()[0])  # push lambda

                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda

        # case: [row M, col +]
        elif input_text[i] == '+':
                stack.append(parsing_table[19][7].split()[0])  # push lambda

                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda

        # case: [row M, col -]
        elif input_text[i] == '-':
                stack.append(parsing_table[19][8].split()[0])  # push lambda

                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda

        # case: [row M, col *]
        elif input_text[i] == '*':
                stack.append(parsing_table[19][9].split()[0])  # push lambda

                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda

        # case: [row M, col /]
        elif input_text[i] == '/':
                stack.append(parsing_table[19][10].split()[0])  # push lambda

                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda
```

```
        # case: [row M, col 0-9]
        elif input_text[i] in digits:
                stack.append(parsing_table[19][11].split()[1])  # push M
                stack.append(parsing_table[19][11].split()[0])  # push D

                print(f"current stack: {stack}")
                current = stack.pop()  # pop D

        else:
                if list(input_text[i - 1])[0] in digits and list(
                                input_text[i - 1])[0] in letters:
                        raise errors.MissingArg(';', i)
                elif input_text[i] == 'display':
                        raise errors.MissingArg(';', i)
                return False

# state U
elif current == 'U':
        # case: [row U, col +]
        if input_text[i] == '+':
                stack.append(parsing_table[20][7].split()[0])  # push +

                print(f"current stack: {stack}")
                current = stack.pop()  # pop +

        # case: [row U, col -]
        elif input_text[i] == '-':
                stack.append(parsing_table[20][8].split()[0])  # push -

                print(f"current stack: {stack}")
                current = stack.pop()  # pop -

        # case: [row U, col 0-9]
        elif input_text[i] in digits:
                stack.append(parsing_table[20][11].split()[0])  # push lambda

                print(f"current stack: {stack}")
                current = stack.pop()  # pop lambda

        else:
                return False

# state D
elif current == 'D':
        # case: [row D, col 0-9]
        if input_text[i] in digits:
                # check which digit is being read
                for digit in digits:
                        if digit == input_text[i]:
                                stack.append(parsing_table[21][int(digit) + 11])  # push digit
                print(f"current stack: {stack}")
                current = stack.pop()  # pop digit
```

```
        else:
                return False

# state L
elif current == 'L':
        if input_text[i] == 'p':
                stack.append(parsing_table[22][21].split()[0])  # push p
                print(f"current stack: {stack}")
                current = stack.pop()

        elif input_text[i] == 'q':
                stack.append(parsing_table[22][22].split()[0])  # push q
                print(f"current stack: {stack}")
                current = stack.pop()

        elif input_text[i] == 'r':
                stack.append(parsing_table[22][23].split()[0])  # push r
                print(f"current stack: {stack}")
                current = stack.pop()

        elif input_text[i] == 's':
                stack.append(parsing_table[22][24].split()[0])  # push s
                print(f"current stack: {stack}")
                current = stack.pop()

        else:
                if input_text[i + 1] in reserved_words:
                        raise errors.MissingArg(';', i)
                return False

# what was popped is what we are reading
elif str(current) == input_text[i]:
        print(f"matched {input_text[i]}")
        print(f"current stack: {stack}")

        # if we are at the end of the file and there were no errors,
        # check arithmetic to prepare for translation to python
        if current == input_text[i] and input_text[i] == 'end.':
                # check for any missing operators, to be able to convert txt to py
                expressions = get_expressions(input_text)
                res = has_operators(expressions, variable_names)

                # if the arithmetic is good, we can conclude check_grammar and return True
                if res:
                        return True
                # otherwise, invalid arithmetic, return False
                else:
                        raise errors.InvalidExpression()

        # if we arent at the end of the file, keep popping and reading
        else:
```

```
                current = stack.pop()  # pop next item
                i += 1

# if the current item that was popped is lambda, pop again
elif current == 'lamb':
        print(f"current stack: {stack}")
        current = stack.pop()

# double check that no errors were missed if we arent in any current state
else:
        if current == 'var':
                raise errors.MissingArg('var', i, inputted_rw[1])
        elif current == 'integer':
                raise errors.MissingArg('integer', i, inputted_rw[2])
        elif current == 'begin':
                raise errors.MissingArg('begin', i, inputted_rw[3])
        elif current == 'display':
                raise errors.MissingArg('display', i, inputted_rw[4])
        elif current == 'end.':
                raise errors.MissingArg('end.', i, inputted_rw[5])
        elif current == '=':
                raise errors.MissingArg('=', i)
        elif current == 'end':
                raise errors.MissingArg('.', i)
        elif current == ';':
                raise errors.MissingArg(';', i)
        elif current == ':':
                raise errors.MissingArg(':', i)
        elif current == ')':
                raise errors.MissingArg(')', i)
        elif current == '(':
                raise errors.MissingArg('(', i)
        elif current == ',':
                raise errors.MissingArg(',', i)

        return False
```

errors.py

```
    class MissingArg(SyntaxError):
        """ Class for any Missing Arguments in the grammar """

        def __init__(self, missing_arg='', position=0, misspelled='', var_name=''):
            """ When errors.MissingArg() is called do the following depending on arguments passed """

            # initialize reserved words and operators
            reserved_words = ['program', 'var', 'begin', 'end.', 'integer', 'display']
            operators = [':', ';', ',', '.', '=', '(', ')']

            # missing an operator
            if missing_arg in operators:
                    print(
                            f"\nSyntaxError: {missing_arg} is missing at position {position} of your file."
                    )
                    exit()

            # either missing or misspelled a reserved word
            elif missing_arg in reserved_words:
                    # if not misspelled, most likely missing:
                    if misspelled == '':
                            print(f"\nSyntaxError: {missing_arg} is expected at position {position} of your
    file.")

                    elif misspelled == missing_arg:
                            print(f"\nSyntaxError: {missing_arg} is expected at position {position} of your
    file.")

                    # if misspelled suggest correct spelling
                    else:
                            print(f"\nThere is no attribute '{misspelled}'. Did you mean: '{missing_arg}' ?")
                    exit()

            # undefined variable
            else:
                    for v in var_name:
                            print(f"\nunknown identifier. variable '{v}' is not defined.")
                    exit()


    class InvalidExpression(ValueError):
        """ Class to for invalid expressions """

        def __init__(self):
            """ When errors.InvalidExpression is passed do the following """

            print("\nInvalidExpression: Arithmetic is invalid. Perhaps you are missing an operator.")
            exit()
```

check_arithmetic.py

```
def has_operators(expression, expected_variables):
    """ Check if a given expression with expected variables is a valid mathematical expression """

    expr_str = '\n'.join(expression)

    with open('math.txt', 'w') as m:
        m.write(expr_str)

    code = compile(expr_str, 'math.py', "exec")

    try:
        exec(code)
        return True
    except:
        return False


def get_expressions(text):
    """ Get all expressions from a given string of text and separate them by white space """

    # initialize
    expressions = []
    stop_reading = False
    operators = ['+', '-', '*', '/', '=', '(', ')']

    # get all expressions in the text
    for word in text:

        # when word is begin, we can grab all expressions
        if word == 'begin':
                k = text.index(word)
                for w in range(k + 1, len(text)):

                        # skip everything to do with display
                        if 'display' in text[w]:
                                stop_reading = True

                        elif ';' in text[w]:
                                stop_reading = False
                                expressions.append(';')
                        elif text[w] == 'end.':
                                break

                        # append everything else to the expressions list
                        elif not stop_reading:
                                expressions.append(text[w])

    # remove all semicolons from the expressions
    expressions = ''.join(expressions)
    expressions = expressions.split(';')
```

```
# remove any extra spaces in the list
while "" in expressions:
    expressions.remove("")

# space out the expressions by operator
spaced_exprs = []

for expr in expressions:
    for op in operators:
            expr = expr.split(op)
            expr = f' {op} '.join(expr)
    spaced_exprs.append(expr)

# return spaced out expressions
return spaced_exprs
```

convertPython.py

```
import sys

def toPython():
    """ Translate the grammar into Python3. """

    # open file for conversion
    with open("finalp2.txt", "r") as file:
        # returns a list of lines
        content = file.readlines()

    #================================================
    # Recognize reserved words and set off flags when found
    # Namely, (program, var, begin, end) for error checking
    program_flag = False
    var_flag = False
    begin_flag = False
    empty_flag = True

    # Iterate through each line
    for w in range(len(content)):

        # Check structure of function with flags
        # two allowed scenarios:
        # 1.) program -> var [var data] -> begin [begin data] -> end
        # 2.) program -> begin [begin data] -> end

        # 'begin' section must not be empty

        # Flags should not be set off if reserved word is used in quotation marks (ie. ("this program is
valid"))

        # Assume "real" reserved words cannot be used in same line as quotations, since they are only allowed
in <stat> -> <write> grammar
        if '"' in content[w]:
                pass
        # Set off flags to distiguish sections
        else:
                # Start of function declaration
                if "program" in content[w]:
                        program_flag = True

                # Start of <dec-list>
                if "var" in content[w]:
                        var_flag = True
                        if program_flag == False:
                                print("error: missing identifier 'program'")
                                sys.exit()

                # Start of <stat-list>
                if "begin" in content[w]:
```

```
                begin_flag = True
                var_flag = False
                if program_flag == False:
                        print("error: missing identifier 'program'")
                        sys.exit()

        # End of function declaration
        if "end" in content[w]:
                if program_flag == False:
                        print("error: missing identifier 'program'")
                        sys.exit()
                if begin_flag == False:
                        print("error: missing identifier 'begin'")
                        sys.exit()
                if empty_flag == True:
                        print("error: empty body is not allowed")
                        sys.exit()

                # set all flags to default if 'end' is found
                program_flag = False
                var_flag = False
                begin_flag = False
                empty_flag = True

# if line is in function declaration, tab the line
if program_flag == True:
        content[w] = "\t" + content[w]

# Now, you can split each line into words
# Identify the object, extract the name, insert into python syntax

# convert line string into array of words
words = content[w].split()

# iterate through list of words
for i in range(len(words)):

        # find function name
        if words[i] == "program":
                global function_name
                function_name = words[i+1]
                #remove semicolon
                function_name = function_name.replace(';', '')
                # insert into python syntax
                content[w] = "def %s():\n" % function_name

        # identifier "var" structure does not exist in python
        # remove var from program
        if words[i] == "var":
                content[w] = ''

        # if in declaration section 'var', assume 'integer' means variable declaration
```

```
                # variable declaration does not exist in python
                # remove variable declaration
                if var_flag == True:
                        if "integer" in content[w]:
                                content[w] = ''

                # 'begin' has no python3 equivalent
                # remove begin
                if "begin" in content[w]:
                        content[w] = ''

                # three scenarios: integer value, math. and display
                # semicolons not needed in any line in python
                if begin_flag == True:
                        content[w] = content[w].replace(';', '')

                        # infer if 'begin' section is empty by checking for <stat-list> indicators
                        if "display" in content[w] or "=" in content[w]:
                                empty_flag = False

                        # Replace the syntax 'display' with python syntax 'print'
                        if "display" in content[w]:
                                content[w] = content[w].replace('display', 'print')

                # Give error if <stat-list> or <dec-list> found before 'begin'
                elif begin_flag == False:
                        if "display" in content[w] or "=" in content[w]:
                                print("error: cannot have statements before identifer 'begin'")
                                sys.exit()

                # end has no python3 equivalent, remove end
                if "end" in content[w]:
                        content[w] = ''

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

    # ensure enough 'end' occurrences to match 'program' occurrences
    if program_flag == True:
        print("error: missing identifier 'end'")
        sys.exit()

#================================================
    # If there are no errors, write to python file
    with open("outputPython.py", "w") as out_file:
        # replace "content" with your new list
        for item in content:
                out_file.write(item)


toPython()

def pull_function():
```

```
return(function_name)
```