Computer Science 323
Fall 2024

Final Project
Jarred Siriban, Ryan Avancena, Kelvin Nguyen

Methods:
CFG, BNF, Remove Left Recursion, First and Follows, Predictive Parsing Table
Languages:
Python

Original Program

Text: “final.txt”

```
program f2024;
(* This program computes and prints the value
of an expression *)
var
  (* declare variables *)
  a ,    b2a ,    c, bba  : integer ;
begin
    a          = 3 ;
    b2a =        14 ;
    c           = 5 ;
  print ( c ); (* display c *)

    (* compute the value of the expression *)
    bba = a1 * ( b2a + 2 * c)     ;
    print ( “value=”,      bba ) ; (* print the value of bba*)

  end
```

Text: “final24.txt”

```
program f2024;
var
a , b2a , c , bba : integer ;
begin
a = 3 ;
b2a = 14 ;
c = 5 ;
print ( c ) ;
bba = ( b2a + 2 * c) * a ;
print ( “Value=” , bba ) ;
end
```

Original Grammar

Original Symbol	RHS
<prog>	→ **program** <identifier>; var <dec-list> **begin** <stat-list> **end**
<identifier>	→ <letter>{ <letter> | <digit> } In EBNF
<dec-list>	→ <dec> : <type>;
<dec>	→ <identifier>, <dec> | <identifier>
<type>	→ **integer**
<stat-list>	→ <stat> | <stat> <stat-list>
<stat>	→ <write> | <assign>
<write>	→ **print** (<str><identifier>);
<str>	→ “value=”, | λ
<assign>	→ <identifier> = <expr>;
<expr>	→ <expr> + <term> | <expr> - <term> | <term>
<term>	→ <term> * <factor> | <term> / <factor> | <factor>
<factor>	→ <identifier> | <number> | (<expr>)
<number>	→ <sign><digit>{<digit>} IN EBNF
<sign>	→ + | - | λ
<digit>	→ 0 | 1 | 2 |... | 9
<letter>	→ a | b | c | d | l | f

Original Grammar and New Abbreviation in BNF

Original Symbol	BNF RHS	Replacement Symbol	BNF Abbr
<prog>	→ **program** <identifier>; **var** <dec-list> **begin** <stat-list> **end**	P	→ **program** I; **var** DL **begin** SL **end**
<identifier>	→ <letter><post-identifier>	I	→ L X
<post-identifier>	→ <letter><post-identifier>	X	→ L X
<post-identifier>	→ <digit><post-identifier>	X	→ DI X
<post-identifier>	→ λ	X	→ λ
<dec-list>	→ <dec> : <type>;	DL	→ D : TY;
<dec>	→ <identifier>, <dec>	D	→ I, D
<dec>	→ <identifier>	D	→ I
<type>	→ **integer**	TY	→ **integer**
<stat-list>	→ <stat>	SL	→ S
<stat-list>	→ <stat><stat-list>	SL	→ S SL
<stat>	→ <write>	S	→ W
<stat>	→ <assign>	S	→ A
<write>	→ **print** (<stat-list><identifier>);	W	→ print (ST I);
<str>	→ “value=”,	ST	→ “value=”,
<str>	→ λ	ST	→ λ
<assign>	→ <identifier> = <expr>;	A	→ I = E;
<expr>	→ <expr> + <term>	E	→ E + T
<expr>	→ <expr> - <term>	E	→ E - T
<expr>	→ <term>	E	→ T
<term>	→ <term> * <factor>	T	→ T * F

<term>	→ <term> / <factor>	T	→ T / F
<term>	→ <factor>	T	→ F
<factor>	→ <identifier>	F	→ I
<factor>	→ <number>	F	→ N
<factor>	→ (<expr>)	F	→ (E)
<number>	→ <sign><digit><post-number>	N	→ SI DI Y
<post-number>	→ <digit><post-number>	Y	→ DI Y
<post-number>	→ λ	Y	→ λ
<sign>	→ +	SI	→ +
<sign>	→ -	SI	→ -
<sign>	→ λ	SI	→ λ
<digit>	→ 0	DI	→ 0
<digit>	→ 1	DI	→ 1
<digit>	→ 2	DI	→ 2
<digit>	→ 3	DI	→ 3
<digit>	→ 4	DI	→ 4
<digit>	→ 5	DI	→ 5
<digit>	→ 6	DI	→ 6
<digit>	→ 7	DI	→ 7
<digit>	→ 8	DI	→ 8
<digit>	→ 9	DI	→ 9
<letter>	→ a	L	→ a
<letter>	→ b	L	→ b
<letter>	→ c	L	→ c
<letter>	→ d	L	→ d

<letter>	→ l	L	→ l
<letter>	→ f	L	→ f

Final BNF Table

Removal of left recursion (Predictive Parsing Table)

Non-Terminals (Replacement Symbol)	Remove Left Recursion
P	→ **program** I; **var** DL **begin** SL **end**
I	→ L X
X	→ L X
X	→ DI X
X	→ λ
DL	→ D : TY;
D	→ I D’
D’	→ , I D’
D’	→ λ
TY	→ **integer**
SL	→ S SL’
SL’	→ SL
SL’	→ λ
S	→ W
S	→ A
W	→ print (ST I);
ST	→ “value=”,
ST	→ λ
A	→ I = E;

E	→ TE’
E’	→ +TE’
E’	→ -TE’
E’	→ λ
T	→ FT’
T’	→ *FT’
T’	→ /FT’
T’	→ λ
F	→ I
F	→ N
F	→ (E)
N	→ SI DI Y
Y	→ DI Y
Y	→ λ
SI	→ +
SI	→ -
SI	→ λ
DI	→ 0
DI	→ 1
DI	→ 2
DI	→ 3
DI	→ 4
DI	→ 5
DI	→ 6
DI	→ 7

DI	→ 8
DI	→ 9
L	→ a
L	→ b
L	→ c
L	→ d
L	→ l
L	→ f

First and Follow Sets

The highlighted rows represent the states that consider all the first and follow sets

Non-Terminals	**Remove Left Recursion**	**First**	**Follow**
P	→ **program** I; **var** DL **begin** SL **end**	program	$
I	→ L X	a b c d l f	; ,) =
X	→ L X	a b c d l f 0 1 2 3 4 5 6 7 8 9 λ	; ,) =
X	→ DI X		
X	→ λ		
DL	→ D : TY;	a b c d l f	begin
D	→ I D’	a b c d l f	:
D’	→ , I D’	, λ	:
D’	→ λ		
TY	→ **integer**	integer	;
SL	→ S SL’	print a b c d l f	end

SL’	→ SL	print a b c d l f λ	end
SL’	→ λ		
S	→ W	print a b c d l f	end print a b c d l f
S	→ A		
W	→ **print** (ST I);	print	end print a b c d l f
ST	→ “value=”,	“value=”, λ	a b c d l f
ST	→ λ		
A	→ I = E;	a b c d l f	end print a b c d l f
E	→ TE’	a b c d l f + - λ (	;)
E’	→ +TE’	+ - λ	;)
E’	→ -TE’		
E’	→ λ		
T	→ FT’	a b c d l f + - λ (	;) + -
T’	→ *FT’	* / λ	;) + -
T’	→ /FT’		
T’	→ λ		
F	→ I	a b c d l f + - λ (	;) + - * /
F	→ N		
F	→ (E)		
N	→ SI DI Y	+ - λ	;) + - * /
Y	→ DI Y	0 1 2 3 4 5 6 7 8 9 λ	;) + - * /
Y	→ λ		
SI	→ +	+ - λ	0 1 2 3 4 5 6 7 8 9
SI	→ -		
SI	→ λ		

DI	→ 0	0 1 2 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9 ;) + - * /
DI	→ 1		
DI	→ 2		
DI	→ 3		
DI	→ 4		
DI	→ 5		
DI	→ 6		
DI	→ 7		
DI	→ 8		
DI	→ 9		
L	→ a	a b c d l f	a b c d l f 0 1 2 3 4 5 6 7 8 9 ; ,) =
L	→ b		
L	→ c		
L	→ d		
L	→ l		
L	→ f		

Predictive Parsing Table

Parsing table split up to four tables due to the size of columns

	a	**b**	**c**	**d**	**l**	**f**
P						
I	LX	LX	LX	LX	LX	LX
X	LX	LX	LX	LX	LX	LX
DL	D : TY;	D : TY;	D : TY;	D : TY;	D : TY;	D : TY;

D	ID’	ID’	ID’	ID’	ID’	ID’
D’						
TY						
SL	S SL’	S SL’	S SL’	S SL’	S SL’	S SL’
SL’						
S	A	A	A	A	A	A
W						
ST	λ	λ	λ	λ	λ	λ
A	I = E;	I = E;	I = E;	I = E;	I = E;	I = E;
E	TE’	TE’	TE’	TE’	TE’	TE’
E’						
T	FT’	FT’	FT’	FT’	FT’	FT’
T’						
F	I	I	I	I	I	I
N						
Y						
SI						
DI						
L	a	b	c	d	l	f

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
P										
I										
X	DI X	DI X	DI X	DI X	DI X	DI X	DI X	DI X	DI X	DI X
DL										

D										
D’										
TY										
SL										
SL’										
S										
W										
ST										
A										
E										
E’										
T										
T’										
F										
N										
Y	DI Y	DI Y	DI Y	DI Y	DI Y	DI Y	DI Y	DI Y	DI Y	DI Y
SI	λ	λ	λ	λ	λ	λ	λ	λ	λ	λ
DI	0	1	2	3	4	5	6	7	8	9
L										

	;	**:**	**,**	**(**	**)**	**=**	**$**	**“value=”,**	**+**	**-**	*****	**/**
P												
I												
X	λ		λ		λ	λ						
DL												

D												
D’		λ	,ID’									
TY												
SL												
SL’												
S												
W												
ST								“value=”,				
A												
E				TE’					TE’	TE’		
E’	λ				λ				+TE’	-TE’		
T												
T’	λ				λ				λ	λ	*FT’	/FT’
F				(E)					N	N		
N									SI DI Y	SI DI Y		
Y	λ				λ				λ	λ	λ	λ
SI									+	-		
DI												
L												

	program	**var**	**begin**	**end**	**integer**	**print**
P	**program** I; **var** DL **begin** SL **end**					
I						

X						
DL						
D						
D’						
TY					**integer**	
SL						S SL’
SL’				λ		SL
S						W
W						print (ST I)
ST						
A						
E						
E’						
T						
T’						
F						
N						
Y						
SI						
DI						
L						

Program

Python file “final24_replicate.py”

```python
a, b2a, c, bba = 3,14,5, None

print(c)
bba = (b2a + 2 * c) * a
print("Value=", bba)
```

Sample run for “final24_replicate.py”

```
PS C:\Users\funko\OneDrive\Documents\GitHub\323_finalproject> python final24_replicate.py
5
Value= 72
PS C:\Users\funko\OneDrive\Documents\GitHub\323_finalproject>
```

Python file “fix_txt.py”

```
'''
ORIGINAL:

program f2024;
(* This program computes and prints the value
of an expression *)
var
(* declare variables *)
a ,    b2a ,    c, bba  : integer ;
begin
    a           = 3 ;
    b2a =        14 ;
    c            = 5 ;
print ( c ); (* display c *)

    (* compute the value of the expression *)
    bba = a1 * ( b2a + 2 * c)     ;
    print ( "value=",      bba ) ; (* print the value of bba*)

end
'''
```

```
content = None

def clean(FILENAME):
    with open(FILENAME, encoding='utf-8') as f:
        content = f.read()

    cleaned_content = []
    inside_comment = False

    for line in content.splitlines():
        line = line.strip()

        # handle multi-line comments
        if "(*" in line and "*)" in line:
            # remove everything between '(*' and '*)' in the same line
            line = line.split("(*")[0] + line.split("*)")[-1]
        elif "(*" in line:
            # start of a multi-line comment
            inside_comment = True
            line = line.split("(*")[0]
        elif "*)" in line:
            # end of a multi-line comment
            inside_comment = False
            line = line.split("*)")[-1]
        elif inside_comment:
            # skip lines inside multi-line comments
            continue

        # process non-empty lines
        if line:
            line = ' '.join(line.split())  # Normalize spaces
            cleaned_content.append(line)

    # join the cleaned lines with a newline
    cleaned_code = "\n".join(cleaned_content)

    # fix spacing around symbols
    cleaned_code = cleaned_code.replace('‟','"')
    cleaned_code = cleaned_code.replace('”','"')
```

```
    cleaned_code = cleaned_code.replace(" ,", ",").replace(', ', ',')
    cleaned_code = cleaned_code.replace(" : ", ":")
    cleaned_code = cleaned_code.replace(" ;", ";")
    cleaned_code = cleaned_code.replace("= ", "=")
    cleaned_code = cleaned_code.replace(" =", "=")
    cleaned_code = cleaned_code.replace("( ", "(").replace(" )", ")")
    cleaned_code = cleaned_code.replace("print (", "print(")


    return cleaned_code

def read(file):
    with open('final.txt', encoding='utf-8') as f:
        content = f.read()
        return content

if __name__ == '__main__':
    content = 'final.txt'
    cleaned_content = clean(content)
    print(cleaned_content)
```

Sample output for “fix_txt.py”

```
PS C:\Users\funko\OneDrive\Documents\GitHub\323_finalproject> python fix_txt.py
program f2024;
var
a,b2a,c,bba:integer;
begin
a=3;
b2a=14;
c=5;
print(c);
bba=a1 * (b2a + 2 * c);
print("value=",bba);
end
```

YOU JUST NEED TO RUN THIS FILE: `final24.py`

Python file “final24.py”

```
from fix_txt import clean
from ppt import valid_input, parse
from handle_identifier import check_identifier
'''

RESERVED WORDS: program, var, end, integer, print

DETECT ERRORS:
- program (if program is spelled wrong)        ... DONE
- var (if var is spelled wrong)               ... DONE
- begin (if begin is spelled wrong)           ... DONE
- integer (if integer is spelled wrong)       ... DONE
- print (if print is spelled wrong)           ... DONE

UNKNOWN IDENTIFIER if variable is not defined:
- ;    (semicolon is missing if grammar required)       ... DONE
- ,    (comma is missing if grammar required)           ... DONE
- .    (period is missing if grammar required)
- (    left parentheses is missing                      ... DONE
- )    right parentheses is missing                     ... DONE

'''


def parse_program_to_arrays(file_content):
    program_array = []
    identifier_array = []
    dec_list_array = []
    stat_list_array = []
    has_semicolon = False
    has_begin = False
    has_end = False
    has_var = False
    current_section = None

    for line in file_content:
        line = line.strip()

        if line.startswith("program"):  # Identify the program and identifier
            parts = line.split()
            program_array.append(parts[0])  # "program"
            identifier_array.append(parts[1].rstrip(";"))  # "f2024"
            has_semicolon = ";" in line  # Check for semicolon

        elif line.startswith("var"):  # Start of declarations
```

```
            has_var = True
            current_section = "dec_list"

        elif line.startswith("begin"):  # Start of statements
            has_begin = True
            current_section = "stat_list"

        elif line.startswith("end"):  # End of the program
            has_end = True
            break

        elif current_section == "dec_list":  # Add declarations
            dec_list_array.append(line)

        elif current_section == "stat_list":  # Add statements
            stat_list_array.append(line.replace(" ", ""))
            # stat_list_array.append(line)

    # Prepare the final arrays with the requested structure
    return {
        "program": program_array,
        "identifier": identifier_array,
        ";": True if has_semicolon else False,
        "var": True if has_var else False,
        "dec_list": dec_list_array,
        "begin": True if has_begin else False,
        "stat_list": stat_list_array,
        "end": True if has_end else False,
    }

READFILE = 'final.txt'

'''

Based on the the notes, the structure of an input should be like this:

    program <identifier> ; var <dec-list> begin <stat-list> end

    example:
        program = ['program']
        identifier = ['f2024']
        ; =
        var = True
        dec_list = ['a , b2a , c, bba : integer ;']
        begin = begin
        stat_list = ['a = 3 ;', 'b2a = 14 ;', 'c = 5 ;', 'print ( c );', 'bba = a1 * ( b2a + 2 * c) ;', 'print (
```

“value=”, bba) ;']
end = end

With that being said. The project functions in a few steps.

1. Cleans the final.txt file and removes comments and indentations, extra spaces, etc .....'cleaned_content'
2. Convert cleaned file into an array ..... 'clean_content_array'
3. Format the array into a dictionary that helps our program interpret it easier ..... 'parsed_content'

i. program : ['program']
ii. identifier : ['f2024']
iii. ; : True
iv. var : True
v. dec_list : ['a, b2a, c, bba : integer;']
vi. begin : True
vii. stat_list : ['a=3;', 'b2a=14;', 'c=5;', 'print (c);', 'bba=a1 * (b2a + 2 * c);', 'print (“value=”, bba);']
viii. end : True

4. This makes sure that the keywords and semicolons are in the proper order even before checking the identifiers in our program.
5. Check identifier, dec_list, and stat_list.

'''

```
def execute_program(READFILE):
    cleaned_content = clean(READFILE)
    cleaned_content_array = cleaned_content.split("\n")
    parsed_content = parse_program_to_arrays(cleaned_content_array)

    # Uncomment the lines below for debugging
    # for k, v in parsed_content.items():
    #     print(f'{k} : {v}')

    is_valid = valid_input(parsed_content)
    final_valid = False

    if is_valid == False:
        print("~ Invalid input format ...")
    else:
        print("~ Valid input format ...")
        final_valid = parse(parsed_content)
        # print(final_valid)
```

```
    if final_valid:
        print("Program is ready to compile!")


execute_program(READFILE)
```

Sample output for “final24.py”

```
proam f2024;
(* This program computes and prints the value
of an expression *)
var
  (* declare variables *)
  a ,    b2a ,    c, bba  : integer ;
begin
    a           = 3 ;
    b2a =        14 ;
    c            = 5 ;
  print ( c ); (* display c *)

    (* compute the value of the expression *)
    bba = a1 * ( b2a + 2 * c )     ;
    print ( “value=”,      bba ) ; (* print the value of bba*)

  end
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   ···   powershell - 323_finalproject
PS C:\Users\jarred\Desktop\CPSC 323\323_finalproject> py final24.py
Error with `program` keyword: missing or empty.
~ Invalid input format ...
PS C:\Users\jarred\Desktop\CPSC 323\323_finalproject>
```

```
1   program f2024;
2   (* This program computes and prints the value
3   of an expression *)
4   var
5     (* declare variables *)
6     a ,    b2a ,    c, bba  : integer ;
7   begin
8       a           = 3 ;
9       b2a =        14 ;
10      c            = 5 ;
11    print ( c ); (* display c *)
12
13      (* compute the value of the expression *)
14      bba = a1 * ( b2a + 2 * c )     ;
15      print ( "value=",      bba ) ; (* print the value of bba*)
16
17    end
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  ···  powershell - 323_finalproject

PS C:\Users\jarred\Desktop\CPSC 323\323_finalproject> py final24.py
~ Valid input format ...
~ Identifier is good! Move to dec_list ...
~ dec_list is good! Move to stat_list ...
~ stat_list is good! Move to end ...
Program is ready to compile!
PS C:\Users\jarred\Desktop\CPSC 323\323_finalproject>
```

```
1   program f2024;
2   (* This program computes and prints the value
3   of an expression *)
4   var
5     (* declare variables *)
6     a ,    b2a ,    c, bba  : integer ;
7   begin
8       a           = 3 ;
9       b2a =        14
10      c           = 5 ;
11    print ( c ); (* display c *)
12
13      (* compute the value of the expression *)
14      bba = a1 * ( b2a + 2 * c )     ;
15      print ( "value=",      bba ) ; (* print the value of bba*)
16
17    end
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  ···  powershell - 323_finalproject

PS C:\Users\jarred\Desktop\CPSC 323\323_finalproject> py final24.py
~ Valid input format ...
~ Identifier is good! Move to dec_list ...
~ dec_list is good! Move to stat_list ...
Missing semicolon and end of assignment.
PS C:\Users\jarred\Desktop\CPSC 323\323_finalproject>
```

Code

final24.py has a few helper functions separated by file

- **check_valid_parse.py**
    - def valid_input(): Checks if the structure of the input is valid after we clean .txt, this effectively checks for any misspellings
    - def parse(): Begins parsing the input if it is valid
- **handle_assign.py**
    - def parse(): Handles the assign operation and the assignment of variables, this function effectively checks if there's a valid identifier on the left side of an '='
- **handle_identifier.py**
    - def check_identifier(): Whenever there's an identifier, we run check_indentifier() to ensure the identifier is valid.
- **handle_print.py**
    - def parse_only_id(): Handles print statements that have no <string>
    - def parse_with_string(): Handles print statements that have <string>
    - def check_string_content(): Ensuring that we have even quotation marks and no missing commas

Python file "check_valid_parse.py"

```
from handle_print import parse_with_string, parse_only_id
from handle_identifier import check_identifier
from handle_assign import handle_assign

def check_dec_list(declarations, type):
    if type != "integer":
        return False
```

```
    # checking the identifiers
    for declaration in declarations:
        valid = check_identifier(declaration)
        if valid:
            continue
        else:
            return False

    return True


def check_stat(stat):
    if len(stat) == 0:
        return False

    if 'print' and "" in stat:
        valid = parse_with_string(stat)
        return True if valid else False
    elif 'print' in stat:
        valid = parse_only_id(stat)
        return True if valid else False
    elif '=' in stat and 'print' not in stat:
        valid = handle_assign(stat)
        return True if valid else False

    else:
        print("Something is likely misspelled.")
        return False


def parse(input_dict):

    ''' we want to focus on identifier, dec_list, and stat_list'''

    for k in input_dict:
        if k == 'identifier':
            for a in input_dict['identifier']:
                valid = check_identifier(a)
                if valid:
```

```
                print("~ Identifier is good! Move to dec_list ...")
                continue
            else:
                return False

        if k == 'dec_list':
            for entry in input_dict['dec_list']:
                # split the declaration by the colon to separate names and type
                parts = entry.split(':')
                if len(parts) == 2:
                    declarations = parts[0].split(',')  # split variable names by commas
                    data_type = parts[1].rstrip(';')   # remove trailing semicolon
                    # print(declarations, data_type)
                    valid = check_dec_list(declarations, data_type)
                    if valid:
                        print("~ dec_list is good! Move to stat_list ...")
                        continue
                    else:
                        print("Error.")
                        return False
                else:
                    print("Error: missing `:`")

        if k == 'stat_list':
            for a in input_dict['stat_list']:
                # print(a)
                valid = check_stat(a)
                # print(f'{a} is {valid}')
                if valid:
                    continue
                else:
                    return False
            print("~ stat_list is good! Move to end ...")

    return True

def valid_input(input_dict):
    if 'program' not in input_dict or not input_dict['program']:
        print('Error with `program` keyword: missing or empty.')
        return False
```

```
    if input_dict['program'][0] != 'program':
        print('Error with `program` keyword: incorrect value.')
        return False


    if 'begin' not in input_dict or not input_dict['begin']:
        print('Error with `begin` keyword.')
        return False

    for reserved in input_dict:
        if not input_dict[reserved]:
            print(f'Missing {reserved}.')
            print(f'If missing is `dec_list`, check spelling of `var`.')
            return False

    if 'end' not in input_dict or not input_dict['end']:
        print('Error with `end` keyword.')
        return False

    return True

if __name__ == '__main__':
    input_string = {
        'program' : ['program'],
        'identifier' : ['f2024'],
        ';' : True,
        'var' : True,
        'dec_list' : ['a,b2a,c,bba:integer;'],
        'begin' : True,
        'stat_list' : ['a=3;', 'b2a=14;', 'c=5;', 'print(c);', 'bba=a1*(b2a+2*c);', 'print("value=",bba);'],
        'end' : True
    }

    is_valid = valid_input(input_string)

    if is_valid == False:
        print("Invalid input.")
    else:
        print("~ Valid Input format!")
        parse(input_string)
```

Python file “handle_assign.py”

```
import re
from handle_identifier import check_identifier

productions = {
   'E': {
      'a': 'TA', 'b': 'TA', 'c': 'TA', 'd': 'TA', 'l': 'TA', 'f': 'TA',
      '+': 'TA', '-': 'TA', '(': 'TA', '0': 'TA', '1': 'TA', '2': 'TA',
      '3': 'TA', '4': 'TA', '5': 'TA', '6': 'TA', '7': 'TA', '8': 'TA',
      '9': 'TA'
   },
   'A': {
      '+': '+TA', '-': '-TA', ')': 'λ', '$': 'λ', ';': 'λ'
   },
   'T': {
      'a': 'FB', 'b': 'FB', 'c': 'FB', 'd': 'FB', 'l': 'FB', 'f': 'FB',
      '+': 'FB', '-': 'FB', '(': 'FB', '0': 'FB', '1': 'FB', '2': 'FB',
      '3': 'FB', '4': 'FB', '5': 'FB', '6': 'FB', '7': 'FB', '8': 'FB',
      '9': 'FB', ';': 'λ'
   },
   'B': {
      '*': '*FB', '/': '/FB', '+': 'λ', '-': 'λ', ')': 'λ', '$': 'λ', ';': 'λ'
   },
   'F': {
      'a': 'I', 'b': 'I', 'c': 'I', 'd': 'I', 'l': 'I', 'f': 'I',
      '+': 'N', '-': 'N', '(': '(E)', '0': 'N', '1': 'N', '2': 'N',
      '3': 'N', '4': 'N', '5': 'N', '6': 'N', '7': 'N', '8': 'N',
      '9': 'N', ';': 'λ'
   },
   'I': {
      'a': 'LX', 'b': 'LX', 'c': 'LX', 'd': 'LX', 'l': 'LX', 'f': 'LX', ';': 'λ'
   },
   'X': {
      'a': 'LX', 'b': 'LX', 'c': 'LX', 'd': 'LX', 'l': 'LX', 'f': 'LX',
      '0': 'DX', '1': 'DX', '2': 'DX', '3': 'DX', '4': 'DX', '5': 'DX',
      '6': 'DX', '7': 'DX', '8': 'DX', '9': 'DX', '+': 'λ', '-': 'λ',
      ')': 'λ', '$': 'λ', ';': 'λ'
   },
```

```
    'N': {
        '+': 'XDY', '-': 'XDY', '0': 'XDY', '1': 'XDY', '2': 'XDY',
        '3': 'XDY', '4': 'XDY', '5': 'XDY', '6': 'XDY', '7': 'XDY',
        '8': 'XDY', '9': 'XDY', ';': 'λ'
    },
    'Y': {
        '0': 'DY', '1': 'DY', '2': 'DY', '3': 'DY', '4': 'DY', '5': 'DY',
        '6': 'DY', '7': 'DY', '8': 'DY', '9': 'DY', '+': 'λ', '-': 'λ',
        ')': 'λ', '$': 'λ'
    },
    'D': {
        '0': '0', '1': '1', '2': '2', '3': '3', '4': '4', '5': '5',
        '6': '6', '7': '7', '8': '8', '9': '9'
    },
    'L': {
        'a': 'a', 'b': 'b', 'c': 'c', 'd': 'd', 'l': 'l', 'f': 'f'
    }
}

def valid_parenthesis(expr):
    stack = []  # Stack to keep track of opening parentheses

    # Dictionary to map closing parentheses to their corresponding opening ones
    matching_parentheses = {')': '(', '}': '{', ']': '['}

    for char in expr:
        # If the character is an opening parenthesis, push it to the stack
        if char in '({[':
            stack.append(char)
        # If the character is a closing parenthesis, check for a match
        elif char in ')}]':
            if not stack or stack[-1] != matching_parentheses[char]:
                return False  # Unmatched or misplaced closing parenthesis
            stack.pop()  # Pop the matching opening parenthesis from the stack

    # If the stack is empty, all parentheses were matched correctly
    return len(stack) == 0

def parse_expr(expr):
    # print(f"\n\n---------- Parsing: {expr} ----------")
```

```
stack = ['$','E']
input_ptr = 0
read = expr[input_ptr]

# loop to iterate through input_string
while len(stack) > 0:
    # print(f"Stack: {stack}")
    popped = stack.pop()
    # print(f"Popped: {popped}")


    # check for the read item and print if the popped item is the same as the read item
    if popped == read:
        # print(f"\t\t\tMatch: [{popped}, {prod}] = {read}")
        input_ptr += 1
        if input_ptr < len(expr):
            read = expr[input_ptr]
        continue

    # look for item based on the predictive parsing table
    if popped in productions and read in productions[popped]:
        prod = productions[popped][read]

        # continue to the next without pushing to stack if lambda
        if prod != 'λ':
            for symbol in prod[::-1]: # push productions in reverse
                stack.append(symbol)

    if popped == '$' and read == ';':
        # print(f"\t\t\tMatch: [End of Input] = {read}")
        break
    else:
        # print(f"No production for [{popped}, {prod}] = {read}")
        continue

# ensure stack is empty and input is completed
# print(stack, input_ptr, len(expr))
if len(stack) == 0 and input_ptr == len(expr)-1:
    # print("Accepted:", expr)
    return True
```

```
    else:
        print("Rejected expression:", expr)
        return False


def handle_assign(assignment):
    if assignment.count('=') != 1:
        return False

    left_identifier, right_side = assignment.split('=')
    left_identifier = left_identifier.strip()
    right_side = right_side.strip()

    left_valid = check_identifier(left_identifier)

    if not left_valid:
        print('Invalid left identifier.')
        return False
    if right_side[-1] != ';':
        print('Missing semicolon and end of assignment.')
        return False
    if not valid_parenthesis(right_side):
        print("Invalid parentheses.")
        return False

    right_valid = parse_expr(right_side)
    if right_valid:
        # print("~ Expression is valid!")
        return True
    else:
        return False
```

Python file “handle_identifier.py”

```
identifier_productions = {
    'I': {  # Start with a letter
        'a': 'LX', 'b': 'LX', 'c': 'LX', 'd': 'LX', 'l': 'LX', 'f': 'LX'
    },
    'X': {  # Continue with letters or digits, or terminate
        'a': 'ZX', 'b': 'ZX', 'c': 'ZX', 'd': 'ZX', 'l': 'ZX', 'f': 'ZX',
```

```
        '0': 'ZX', '1': 'ZX', '2': 'ZX', '3': 'ZX', '4': 'ZX', '5': 'ZX',
        '6': 'ZX', '7': 'ZX', '8': 'ZX', '9': 'ZX', ';': 'λ', '=': 'λ', '$': 'λ'
    },
    'L': {  # Match letters
        'a': 'a', 'b': 'b', 'c': 'c', 'd': 'd', 'l': 'l', 'f': 'f'
    },
    'Z': {  # Match digits
        '0': '0', '1': '1', '2': '2', '3': '3', '4': '4',
        '5': '5', '6': '6', '7': '7', '8': '8', '9': '9',
        'a': 'a', 'b': 'b', 'c': 'c', 'd': 'd', 'l': 'l', 'f': 'f'
    },
}


def check_identifier(line):
    if len(line) == 0:
        return False

    stack = ['$','I']  # Start with 'I' for the first letter
    input_ptr = 0
    read = line[input_ptr]

    while len(stack) > 0:
        # print(f"Stack: {stack}")
        popped = stack.pop()
        # print(f"Popped: {popped}")

        # Match terminal symbol
        if popped == read:
            # print(f"\t\t\tMatch: {popped} == {read}")
            input_ptr += 1
            if input_ptr < len(line):
                read = line[input_ptr]
            else:
                read = '$'  # End of input marker
            continue

        # Handle non-terminal symbol
        if popped in identifier_productions:
            if read in identifier_productions[popped]:
```

```
            prod = identifier_productions[popped][read]
            # print(f"Applying Production: {popped} -> {prod}")

            # Handle lambda
            if prod == 'λ':
                continue

            # Push production in reverse order
            for symbol in reversed(prod):
                stack.append(symbol)
        else:
            # print(f"No production for [{popped}, {read}]")
            print(f"Issue with identifier: {line}.")
            return False
    else:
        print(f"Issue with identifier: {line}.")
        return False

# Final check for acceptance
if len(stack) == 0 and read == '$':
    # print("Accepted:", line)
    return True
else:
    # print("Rejected:", line)
    return False
```

Python file “handle_print.py”

```
'''

write: print(identifier)    ... DONE
write: print("string", identifier)    ... NEED TO DO

'''

import re
from handle_identifier import check_identifier

def parse_only_id(stat):
```

```
    # Use regex to split by '(', ')', and ';', but keep them in the result
    tokens = re.findall(r'print|[();]|\w+', stat)
    # print(tokens)
    if 'print' in tokens:
        print_index = tokens.index('print')
        if print_index + 1 >= len(tokens) or tokens[print_index + 1] != '(':
            return False  # 'print' is not followed by '('
    else:
        return False

    if ')' in tokens:
        close_paren_index = tokens.index(')')
        if close_paren_index + 1 >= len(tokens) or tokens[close_paren_index + 1] != ';':
            return False  # ')' is not followed by ';'

    if '(' in tokens and ')' in tokens:
        start_index = tokens.index('(')
        end_index = tokens.index(')')
        content_inside_parentheses = tokens[start_index + 1:end_index]

        valid_identifier = check_identifier(content_inside_parentheses[0])
        if valid_identifier:
            return True
        else:
            return False
    else:
        return False

def check_string_content(content):
    """Check that the string content inside quotation marks is valid."""
    # Join the list into a single string
    content_str = ''.join(content)

    if '"' not in content_str or content_str.count('"') != 2:
        print("Invalid syntax: Missing or unbalanced quotation marks")
        return False

    # Extract content between quotation marks
    quote_start = content_str.index('"')
    quote_end = content_str.index('"', quote_start + 1)
```

```
    # Include everything between and within the quotation marks
    quoted_string = content_str[quote_start + 1:quote_end]

    if not quoted_string.isprintable():
        print("Invalid string inside quotation marks")
        return False

    # Ensure there is a comma right after the closing quotation mark
    if quote_end + 1 >= len(content_str) or content_str[quote_end + 1] != ",":
        print("Invalid syntax: Missing comma after closing quotation mark")
        return False

    after_comma = content_str[quote_end + 2:].strip()
    if not check_identifier(after_comma):
        return False

    # print(f"Valid string content: {quoted_string}")
    return True

def parse_with_string(stat):
    """Parse the statement and validate the structure."""
    tokens = re.findall(r'print|[();,"]|(?:\w+|=)', stat)

    # Check for 'print' followed by '('
    if 'print' in tokens:
        print_index = tokens.index('print')
        if print_index + 1 >= len(tokens) or tokens[print_index + 1] != '(':
            print("Invalid syntax: 'print' is not followed by '('")
            return False
    else:
        print("Invalid syntax: Missing 'print'")
        return False

    # Check for ')' followed by ';'
    if ')' in tokens:
        close_paren_index = tokens.index(')')
        if close_paren_index + 1 >= len(tokens) or tokens[close_paren_index + 1] != ';':
            print("Invalid syntax: ')' is not followed by ';'")
            return False
```

```
    else:
        print("Invalid syntax: Missing ')'")
        return False

    # Extract and validate content inside parentheses
    if '(' in tokens and ')' in tokens:
        start_index = tokens.index('(')
        end_index = tokens.index(')')
        content_inside_parentheses = tokens[start_index + 1:end_index]
        is_valid = check_string_content(content_inside_parentheses)
        return is_valid
    else:
        print("Invalid syntax: Missing '(' or ')'")
        return False
```
