Queens College of CUNY
Department of Computer Science
Programming Languages
(CSCI 316)
Winter 2026

Assignment #6
"Assignments and Statements"
Due: January 11, 2026

Introduction:

In this assignment, we explore Python code to parse and process arithmetic expressions and assignment statements. We utilize the well-known Python "PLY", Python LEX-YACC, where LEX is a Lexical Analyzer that divides the input text (basic program) into a sequence of tokens based on regular expression rules, while YACC (Yet Another Compiler Compiler) takes those tokens and organizes them into a meaningful structure (like a parse tree) according to a specified grammar.

Submissions:

In the Google form, please submit:

- Assignment06.py (source code)
- Assignment06.txt (console output)

Tasks:

[0] Create a new program Assignment06.py, update the assignment name and number in the comment at the top, and perform any other preliminary tasks associated with a new assignment.

[1] Use the following code and get it to run for the provided demo data and review the output

[2] Define a new binary operator & such that x & y = max(x, y). Make it have even lower priority than addition (+) and subtraction (-). Modify the code below to support this operator. Recommendation: search the code for + and PLUS to see what is done for addition, and add the appropriate code for & and MAX. Test with a few expressions that use this operator.

[3] Define a new binary operator $ such that x $ y = min(x, y). Make it have even lower priority than addition (+) and subtraction (-). See step [2] for suggestions.

```
"""
PLY example: arithmetic expressions + statements
Now supports newline-terminated statements (no semicolons required).

Statements may end with:
  - NEWLINE
  - SEMI-COLON
Blank lines are allowed.

Install:
  pip install ply
Run:
  python ply_calc_newline.py
"""

import ply.lex as lex
```

```
import ply.yacc as yacc

# ----------------------------
# LEXER
# ----------------------------

DEBUG_LEX = False

reserved = {"print": "PRINT"}

tokens = (
    "ID",
    "NUMBER",
    "PLUS",
    "MINUS",
    "TIMES",
    "DIVIDE",
    "MOD",
    "POW",
    "EQUALS",
    "LPAREN",
    "RPAREN",
    "SEMI",
    "NEWLINE",
) + tuple(reserved.values())

t_PLUS   = r"\+"
t_MINUS  = r"-"
t_TIMES  = r"\*"
t_DIVIDE = r"/"
t_MOD    = r"%"
t_POW    = r"\^"
t_EQUALS = r"="
t_LPAREN = r"\("
t_RPAREN = r"\)"
t_SEMI   = r";"

# Ignore spaces/tabs, but NOT newlines
t_ignore = " \t\r"

def t_NUMBER(t):
    r"(\d+(\.\d*)?|\.\d+)"
    t.value = int(t.value) if "." not in t.value else float(t.value)
    return t

def t_ID(t):
    r"[A-Za-z_][A-Za-z0-9_]*"
    t.type = reserved.get(t.value, "ID")
    if DEBUG_LEX:
        print(f"TOKEN {t.type}({t.value}) at line {t.lineno}")
    return t



def t_NEWLINE(t):
```

```
    r"\n+"
    t.lexer.lineno += len(t.value)
    # Collapse consecutive newlines into a single NEWLINE token
    t.value = "\n"
    if DEBUG_LEX:
        print(f"TOKEN {t.type}({t.value}) at line {t.lineno}")
    return t

def t_error(t):
    raise SyntaxError(f"Illegal character {t.value[0]!r} at line {t.lexer.lineno}")

lexer = lex.lex()

# ----------------------------
# PARSER
# ----------------------------

precedence = (
    ("left", "PLUS", "MINUS"),
    ("left", "TIMES", "DIVIDE", "MOD"),
    ("right", "POW"),
    ("right", "UMINUS"),
)

# AST shapes:
# ("program", [stmt...])
# ("assign", name, expr)
# ("exprstmt", expr)
# ("print", expr)
# expr nodes: ("num", v), ("var", name), ("unop","-",e), ("binop",op,l,r)

def p_program_empty(p):
    """program : opt_newlines"""
    p[0] = ("program", [])

def p_program_append(p):
    """program : program opt_newlines statement"""
    p[0] = ("program", p[1][1] + [p[3]])

def p_opt_newlines_empty(p):
    """opt_newlines : """
    p[0] = None

def p_opt_newlines_some(p):
    """opt_newlines : opt_newlines NEWLINE"""
    p[0] = None

def p_end(p):
    """end : NEWLINE
           | SEMI"""
    p[0] = None

def p_statement_assign(p):
    """statement : ID EQUALS expr end"""
    p[0] = ("assign", p[1], p[3])
```

```
def p_statement_expr(p):
    """statement : expr end"""
    p[0] = ("exprstmt", p[1])

def p_statement_print(p):
    """statement : PRINT LPAREN expr RPAREN end"""
    p[0] = ("print", p[3])

def p_expr_binop(p):
    """expr : expr PLUS expr
            | expr MINUS expr
            | expr TIMES expr
            | expr DIVIDE expr
            | expr MOD expr
            | expr POW expr"""
    p[0] = ("binop", p[2], p[1], p[3])

def p_expr_uminus(p):
    """expr : MINUS expr %prec UMINUS"""
    p[0] = ("unop", "-", p[2])

def p_expr_group(p):
    """expr : LPAREN expr RPAREN"""
    p[0] = p[2]

def p_expr_number(p):
    """expr : NUMBER"""
    p[0] = ("num", p[1])

def p_expr_var(p):
    """expr : ID"""
    p[0] = ("var", p[1])

def p_error(p):
    if p is None:
        return  # allow clean EOF
    raise SyntaxError(f"Syntax error near {p.value!r} at line {p.lineno}")

parser = yacc.yacc(start="program")

# ----------------------------
# EVALUATOR
# ----------------------------

class EvalError(Exception):
    pass

def eval_expr(node, env):
    kind = node[0]
    if kind == "num":
        return node[1]
    if kind == "var":
        name = node[1]
        if name not in env:
```

```
            raise EvalError(f"Undefined variable: {name}")
        return env[name]
    if kind == "unop":
        _, op, e = node
        v = eval_expr(e, env)
        if op == "-":
            return -v
        raise EvalError(f"Unknown unary op: {op}")
    if kind == "binop":
        _, op, l, r = node
        a = eval_expr(l, env)
        b = eval_expr(r, env)
        if op == "+": return a + b
        if op == "-": return a - b
        if op == "*": return a * b
        if op == "/": return a / b
        if op == "%": return a % b
        if op == "^": return a ** b
        raise EvalError(f"Unknown binop: {op}")
    raise EvalError(f"Unknown expr node: {node}")

def exec_program(ast, env=None):
    if env is None:
        env = {}
    assert ast[0] == "program"
    for stmt in ast[1]:
        k = stmt[0]
        if k == "assign":
            _, name, e = stmt
            env[name] = eval_expr(e, env)
        elif k == "exprstmt":
            _, e = stmt
            _ = eval_expr(e, env)
        elif k == "print":
            _, e = stmt
            print(eval_expr(e, env))
        else:
            raise EvalError(f"Unknown statement: {stmt}")
    return env

def dump_tokens(text):
    lexer.input(text)
    print("LEXEMES / TOKENS:")
    while True:
        tok = lexer.token()
        if not tok:
            break
        print(f"{tok.type:<8} value={tok.value!r:<10} line={tok.lineno}")

# ----------------------------
# DEMO / REPL
# ----------------------------

if __name__ == "__main__":
    sample = """\
```

```
x = 2 + 3 * 4
y = x ^ 2 - 10
print(y)
print(-(1 + 2) * 3)

z = y % 7
print(z)
"""
    dump_tokens(sample)# optional to see tokens
    # parser = yacc.yacc(start="program", debug=True)
    ast = parser.parse(sample, lexer=lexer, debug=False)
    print("AST:", ast)
    env = exec_program(ast)
    print("Final env:", env)

    print("\nREPL: press Enter on a blank line to execute buffered input (Ctrl+C to quit).")
    buf = []
    try:
        while True:
            line = input(">>> ")
            if line.strip() == "":
                if buf:
                    text = "\n".join(buf) + "\n"
                    tree = parser.parse(text, lexer=lexer)
                    exec_program(tree, env)
                    buf.clear()
                continue
            buf.append(line)
    except (KeyboardInterrupt, EOFError):
        print("\nbye")
```