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

Assignment #4
"Context-Free Grammars and Derivations"
Due: January 9, 2026

Introduction:

In this assignment, we use Python code to simulate a context-free grammar (CFG), produce a leftmost derivation, and output the associated parse tree for small Python programs.

Submissions:

In the Google form, please submit:

- Assignment04.py (source code)
- Assignment04.txt (console output)

Tasks:

[0] Create a new program Assignment04.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 obtained via "vibe programming".

[2] Run CFGDriver for the three given one-line Python programs and two more that you construct

[3] Understand the output of the simulations. (While you would not be expected to write such code on an exam, you need to be able to use a CFG to produce a derivation.)

```
from dataclasses import dataclass
from typing import List, Tuple, Dict, Optional, Union

Symbol = str

@dataclass
class Node:
    sym: Symbol
    children: List["Node"]

    def pretty(self, indent: str = "", last: bool = True) -> str:
        elbow = "└─" if last else "├─"
        out = f"{indent}{elbow}{self.sym}\n"
        indent2 = indent + ("  " if last else "│ ")
        for i, ch in enumerate(self.children):
            out += ch.pretty(indent2, i == len(self.children) - 1)
        return out


class CFGDeriver:
```

```
def __init__(self):
    self.start: Symbol = "Stmt"
    self.nonterminals = {"Stmt", "Expr", "ExprP", "Term", "TermP", "Factor"}

    # Productions are lists of alternatives; each alternative is a list of symbols.
    self.prods: Dict[Symbol, List[List[Symbol]]] = {
        "Stmt": [["ID", "=", "Expr"]],
        "Expr": [["Term", "ExprP"]],
        "ExprP": [["+", "Term", "ExprP"], ["-", "Term", "ExprP"], ["ε"]],
        "Term": [["Factor", "TermP"]],
        "TermP": [["*", "Factor", "TermP"], ["/", "Factor", "TermP"], ["ε"]],
        "Factor": [["(", "Expr", ")"], ["ID"], ["NUM"]],
    }

def tokenize(self, s: str) -> List[str]:
    """
    Very small tokenizer:
      - identifiers: letters/underscore followed by letters/digits/underscore
      - numbers: digits
      - symbols: = + - * / ( )
    """
    import re
    token_spec = [
        ("NUM",   r"\d+"),
        ("ID",    r"[A-Za-z_]\w*"),
        ("SKIP",  r"[ \t]+"),
        ("SYMS",  r"==|!=|<=|>=|[=+\-*/()]"),  # keep simple
        ("MISM",  r"."),
    ]
    tok_re = "|".join(f"(?P<{name}>{pat})" for name, pat in token_spec)
    out = []
    for m in re.finditer(tok_re, s):
        kind = m.lastgroup
        val = m.group()
        if kind == "SKIP":
            continue
        if kind == "MISM":
            raise ValueError(f"Unexpected character: {val!r}")
        if kind in ("ID", "NUM"):
            out.append(kind)      # normalize to token type
            out.append(val)       # and keep lexeme for printing
        else:
            out.append(val)
    # For parsing we want a stream of *terminals*; keep lexemes in a parallel list.
    # We'll compress to terminals where ID/NUM match by type but record actual words.
    terminals = []
    i = 0
    while i < len(out):
        if out[i] in ("ID", "NUM"):
            terminals.append(out[i])  # terminal is token type
            i += 2
        else:
            terminals.append(out[i])
            i += 1
```

```
    return terminals

def derive(self, program: str) -> None:
    tokens = self.tokenize(program)
    print("Input:", program)
    print("Tokens:", tokens)

    derivation: List[Tuple[Symbol, List[Symbol]]] = []  # (LHS, RHS chosen)
    tree = self._parse_symbol(self.start, tokens, 0, derivation)

    if tree is None:
        print("\nParse failed: string not generated by the grammar.")
        return

    node, pos = tree
    if pos != len(tokens):
        print("\nParse stopped early; remaining tokens:", tokens[pos:])
        return

    print("\nProductions used (in order):")
    for lhs, rhs in derivation:
        print(f"  {lhs} -> {' '.join(rhs)}")

    print("\nLeftmost derivation (sentential forms):")
    forms = self._sentential_forms(derivation)
    for i, f in enumerate(forms):
        print(f"{i:2d}: {f}")

    print("\nParse tree:")
    print(node.pretty())

def _sentential_forms(self, derivation: List[Tuple[Symbol, List[Symbol]]]) -> List[str]:
    """
    Reconstruct a leftmost derivation from recorded production applications.
    We assume our parser expands the leftmost pending nonterminal at each step
    (true for this top-down approach).
    """
    sent = [self.start]
    forms = [" ".join(sent)]
    for lhs, rhs in derivation:
        # find leftmost occurrence of lhs in sentential form
        try:
            idx = sent.index(lhs)
        except ValueError:
            # if not found (shouldn't happen), skip
            continue
        replacement = [] if rhs == ["ε"] else rhs
        sent = sent[:idx] + replacement + sent[idx + 1:]
        forms.append(" ".join(sent) if sent else "ε")
    return forms

def _parse_symbol(
    self,
    sym: Symbol,
    tokens: List[str],
```

```
        pos: int,
        derivation: List[Tuple[Symbol, List[Symbol]]],
    ) -> Optional[Tuple[Node, int]]:
        # Terminal
        if sym not in self.nonterminals:
            if sym == "ε":
                return Node("ε", []), pos
            if pos < len(tokens) and tokens[pos] == sym:
                return Node(sym, []), pos + 1
            return None

        # Nonterminal: try alternatives
        for rhs in self.prods[sym]:
            # Record attempt; only commit if this alternative succeeds.
            saved_len = len(derivation)
            saved_pos = pos
            children: List[Node] = []

            # Commit production choice now, but roll back if it fails.
            derivation.append((sym, rhs))

            ok = True
            cur_pos = pos
            for part in rhs:
                res = self._parse_symbol(part, tokens, cur_pos, derivation)
                if res is None:
                    ok = False
                    break
                child, cur_pos = res
                # omit epsilon nodes from tree display if you prefer; keep them for clarity
                children.append(child)

            if ok:
                return Node(sym, children), cur_pos

            # rollback
            derivation[:] = derivation[:saved_len]
            pos = saved_pos

        return None

"""
    Simple CFG derivation simulator + backtracking parser.

    Grammar (Python-like subset):
      Stmt   -> ID '=' Expr
      Expr   -> Term ExprP
      ExprP  -> '+' Term ExprP | '-' Term ExprP | ε
      Term   -> Factor TermP
      TermP  -> '*' Factor TermP | '/' Factor TermP | ε
      Factor -> '(' Expr ')' | ID | NUM
"""

def main():
    g = CFGDeriver()
```

```
    g.derive("x = 1 + 2 * 3")
    g.derive("total = ( 1 + 2 ) * 3")
    g.derive("y = a / ( b - 2 )")


if __name__ == "__main__":
    main()
```