CS3372 / Assignment 1

Abstract

This report presents a context-free grammar (CFG) for a simple Python-like language and verifies that it generates the program required by Assignment 1. The grammar divides the syntax into clear components, from general statements to basic tokens. To prove that the required program belongs to the generated language, a leftmost derivation is constructed. Furthermore, a lightweight parser is used for experimental testing; it accepts the correct program and rejects invalid cases. Both the mathematical derivation and the program implementation demonstrate that the grammar meets the assignment requirements.

1 Problem Statement

This assignment focuses on designing a CFG for a simplified Python-like language capable of generating a specific sequence of function definitions and conditional statements. The primary objective is to formally derive the target program from the designed grammar and present the methodology and experimental evidence. The required concrete program is:

```
def add(x, y): z = x + y return z end
if 10 > 5: a = 1 else: a = 2 end
```

The report constructs a small language that contains the required program and has a sufficiently clear structure for formal verification.

2 Background

A context-free grammar is formally represented as a four-tuple

$$G = (V, \Sigma, R, S),$$

where V is a finite set of variables (nonterminals), Σ is a finite set of terminals disjoint from V, R is a finite set of production rules, and S is the start variable [1]. This is consistent with standard textbook definitions of CFGs [2, 3].

A derivation begins at the start variable and repeatedly replaces a nonterminal using a production rule until only terminals remain. Following the formalisms introduced in the course materials, this sequence of rule substitutions is termed a derivation [1]. This report specifically employs a *leftmost derivation*, where the leftmost nonterminal is systematically expanded at each step. Linz and Rodger formally define a leftmost derivation in exactly this manner [3]. Context-free grammars are particularly instrumental in describing programming-language syntax and facilitating parsing mechanisms [2, 4].

3 Design Progress

The grammar was iteratively developed through the following methodology:

1. **Identify top-level constructs.** The sample consists of a function definition followed by an `if-else` statement. A general statement-list nonterminal therefore becomes the top-level structure.

2. **Separate statement forms.** Function definitions, conditionals, and assignments are represented by different nonterminals. This follows the common CFG design idea of generating independent fixed-order regions through separate variables [4].

3. **Factor expressions and conditions.** Arithmetic addition and the comparison operator `>` are represented separately so that the function body and conditional guard have clear syntactic roles.

4. **Add lexical structure.** Identifiers are generated from lowercase letters and digits, and numbers are generated from digits. This makes it possible to derive the concrete lexemes `add`, `x`, `y`, `z`, `a`, `10`, `5`, `1`, and `2` rather than leaving them as unexplained placeholders.

5. **Verify membership.** A leftmost derivation is constructed for the required program. A small parser is also run on the required program and malformed variants as an experimental check.

4 Grammar Design

4.1 Language Specification

The language is intentionally small. It supports:

- one or more top-level statements;
- function definitions with one or more parameters;
- a function body containing one assignment followed by one return statement;
- `if-else` statements with one assignment in each branch;
- assignments;
- operands that are identifiers or nonnegative decimal integers;
- either a single operand or one binary addition in an expression; and
- a comparison of the form `operand > operand`.

Whitespace and line breaks are treated as separators for readability and are not significant to the syntactic structure. Reserved keywords and punctuation are written as terminal symbols.

4.2 Formal CFG

Let

$$G = (V, \Sigma, R, S),$$

with start symbol

$$S = \langle Program \rangle.$$

The nonterminal set is

$$V = \{\langle Program \rangle, \langle StatementList \rangle, \langle Statement \rangle, \langle FunctionDef \rangle, \langle ParamList \rangle, \langle ParamTail \rangle, \langle IfStmt \rangle, \langle Condition \rangle, \langle Assignment \rangle, \langle Expression \rangle, \langle Operand \rangle, \langle Identifier \rangle, \langle IdTail \rangle, \langle Number \rangle, \langle NumTail \rangle, \langle Letter \rangle, \langle Digit \rangle\}.$$

The terminal alphabet contains the reserved words and punctuation used by the language, together with lowercase letters and decimal digits:

$$\Sigma = \{\texttt{def}, \texttt{return}, \texttt{end}, \texttt{if}, \texttt{else}, \texttt{(}, \texttt{)}, \texttt{,}, \texttt{:}, \texttt{=}, \texttt{+}, \texttt{>}, \texttt{a}, \ldots, \texttt{z}, \texttt{0}, \ldots, \texttt{9}\}.$$

Here the reserved words are treated as atomic lexical terminals, while identifier and number lexemes are generated from letter and digit terminals. This is a convenient presentation convention for a programming-language grammar.

The production set R is:

Table 1: Production rules of the CFG.

Nonterminal		Production
⟨*Program*⟩	→	⟨*StatementList*⟩
⟨*StatementList*⟩	→	⟨*Statement*⟩ ⟨*StatementList*⟩ | ⟨*Statement*⟩
⟨*Statement*⟩	→	⟨*FunctionDef*⟩ | ⟨*IfStmt*⟩ | ⟨*Assignment*⟩
⟨*FunctionDef*⟩	→	`def` ⟨*Identifier*⟩ `(` ⟨*ParamList*⟩ `)` `:` ⟨*Assignment*⟩ `return` ⟨*Identifier*⟩ `end`
⟨*ParamList*⟩	→	⟨*Identifier*⟩ ⟨*ParamTail*⟩
⟨*ParamTail*⟩	→	`,` ⟨*Identifier*⟩ ⟨*ParamTail*⟩ | ϵ
⟨*IfStmt*⟩	→	`if` ⟨*Condition*⟩ `:` ⟨*Assignment*⟩ `else` `:` ⟨*Assignment*⟩ `end`
⟨*Condition*⟩	→	⟨*Operand*⟩ `>` ⟨*Operand*⟩
⟨*Assignment*⟩	→	⟨*Identifier*⟩ `=` ⟨*Expression*⟩
⟨*Expression*⟩	→	⟨*Operand*⟩ `+` ⟨*Operand*⟩ | ⟨*Operand*⟩
⟨*Operand*⟩	→	⟨*Identifier*⟩ | ⟨*Number*⟩
⟨*Identifier*⟩	→	⟨*Letter*⟩ ⟨*IdTail*⟩
⟨*IdTail*⟩	→	⟨*Letter*⟩ ⟨*IdTail*⟩ | ⟨*Digit*⟩ ⟨*IdTail*⟩ | ϵ
⟨*Number*⟩	→	⟨*Digit*⟩ ⟨*NumTail*⟩
⟨*NumTail*⟩	→	⟨*Digit*⟩ ⟨*NumTail*⟩ | ϵ
⟨*Letter*⟩	→	`a` | `b` | $\cdots$ | `z`
⟨*Digit*⟩	→	`0` | `1` | $\cdots$ | `9`

Every production has exactly one nonterminal on its left-hand side, so the grammar satisfies the context-free form $A \to x$, where A is a variable and x is a string of variables and terminals [3].

4.3 Design Approach

The main structural rule is

$$\langle \textit{StatementList} \rangle \to \langle \textit{Statement} \rangle \ \langle \textit{StatementList} \rangle \mid \langle \textit{Statement} \rangle,$$

which allows a program to contain one or more top-level statements. The required program therefore consists of two statements: a ⟨*FunctionDef*⟩ followed by an ⟨*IfStmt*⟩.

The function rule fixes the order of its components: keyword, function name, parameter list, colon, assignment, return statement, and `end`. Likewise, the conditional rule fixes the order `if condition : assignment else : assignment end`. These constraints guarantee that the required punctuation and block delimiters appear in the correct positions.

The recursive lexical rules permit identifiers and numbers of arbitrary positive length. For example, ⟨*IdTail*⟩ can repeatedly add letters or digits and then terminate with ϵ. Similarly, ⟨*NumTail*⟩ can repeatedly add digits and then terminate.

Furthermore, the explicit `end` keyword gives every conditional a clear closing boundary and avoids the classic "dangling else" problem encountered in less constrained grammars. Some language grammars instead rely on matched/unmatched statement rules or indentation tokens such as `INDENT` and `DEDENT` to associate an `else` clause with the correct `if`. In this restricted language, the fixed sequence `if ... : ... else : ... end` makes that association explicit and yields an unambiguous conditional structure.

5 Derivation of the Required Program

5.1 Lexical subderivations

To keep the main derivation readable, repeated lexical expansions are abbreviated by $\overset{L}{\Rightarrow}^*$. Each occurrence represents the corresponding finite sequence of ordinary leftmost derivation steps using the lexical productions defined in Section 4.2; examples of those steps are given below.

$$\begin{aligned}
\langle \mathit{Identifier} \rangle &\Rightarrow \langle \mathit{Letter} \rangle \langle \mathit{IdTail} \rangle \Rightarrow \texttt{a}\langle \mathit{IdTail} \rangle \Rightarrow \texttt{a}\langle \mathit{Letter} \rangle \langle \mathit{IdTail} \rangle \Rightarrow \texttt{ad}\langle \mathit{IdTail} \rangle \\
&\Rightarrow \texttt{ad}\langle \mathit{Letter} \rangle \langle \mathit{IdTail} \rangle \Rightarrow \texttt{add}\langle \mathit{IdTail} \rangle \Rightarrow \texttt{add}, \\
\langle \mathit{Identifier} \rangle &\Rightarrow \langle \mathit{Letter} \rangle \langle \mathit{IdTail} \rangle \Rightarrow \texttt{x}\langle \mathit{IdTail} \rangle \Rightarrow \texttt{x}, \\
\langle \mathit{Number} \rangle &\Rightarrow \langle \mathit{Digit} \rangle \langle \mathit{NumTail} \rangle \Rightarrow \texttt{1}\langle \mathit{NumTail} \rangle \Rightarrow \texttt{1}\langle \mathit{Digit} \rangle \langle \mathit{NumTail} \rangle \Rightarrow \texttt{10}\langle \mathit{NumTail} \rangle \Rightarrow \texttt{10}, \\
\langle \mathit{Number} \rangle &\Rightarrow \langle \mathit{Digit} \rangle \langle \mathit{NumTail} \rangle \Rightarrow \texttt{5}\langle \mathit{NumTail} \rangle \Rightarrow \texttt{5}.
\end{aligned}$$

The same one-letter identifier pattern derives `y`, `z`, and `a`; the same one-digit number pattern derives `1` and `2`.

5.2 Leftmost derivation

Let

$$\begin{aligned}
f &= \texttt{def add(x, y): z = x + y return z end}, \\
i &= \texttt{if 10 > 5: a = 1 else: a = 2 end}.
\end{aligned}$$

The complete program is $w = fi$ (with whitespace between the two top-level statements). A leftmost derivation at the program level is

$$\begin{aligned}
\langle \mathit{Program} \rangle &\Rightarrow \langle \mathit{StatementList} \rangle \\
&\Rightarrow \langle \mathit{Statement} \rangle \ \langle \mathit{StatementList} \rangle \\
&\Rightarrow \langle \mathit{FunctionDef} \rangle \ \langle \mathit{StatementList} \rangle \\
&\overset{L}{\Rightarrow}^* f \ \langle \mathit{StatementList} \rangle \\
&\Rightarrow f \ \langle \mathit{Statement} \rangle \\
&\Rightarrow f \ \langle \mathit{IfStmt} \rangle \\
&\overset{L}{\Rightarrow}^* fi = w.
\end{aligned}$$

This leftmost derivation systematically expands the nonterminals until the required program is obtained. The two starred parts are expanded below. Because $\langle \mathit{FunctionDef} \rangle$ is expanded completely before the $\langle \mathit{StatementList} \rangle$ to its right, and each following subderivation also expands its leftmost remaining nonterminal, the complete sequence is a leftmost derivation.

5.2.1 Leftmost expansion of $\langle \mathit{FunctionDef} \rangle$

Table 2: Leftmost expansion of $\langle \mathit{FunctionDef} \rangle$.

Stage	Sentential form
0	$\langle \mathit{FunctionDef} \rangle$
1	$\Rightarrow$ `def` $\langle \mathit{Identifier} \rangle$`(`$\langle \mathit{ParamList} \rangle$`):` $\langle \mathit{Assignment} \rangle$ `return` $\langle \mathit{Identifier} \rangle$ `end`
2	$\overset{L}{\Rightarrow}^*$ `def add(`$\langle \mathit{ParamList} \rangle$`):` $\langle \mathit{Assignment} \rangle$ `return` $\langle \mathit{Identifier} \rangle$ `end`

Stage	Sentential form
3	$\Rightarrow$ `def add(`⟨*Identifier*⟩ ⟨*ParamTail*⟩`):` ⟨*Assignment*⟩ `return` ⟨*Identifier*⟩ `end`
4	$\overset{L}{\Rightarrow}^*$ `def add(x` ⟨*ParamTail*⟩`):` ⟨*Assignment*⟩ `return` ⟨*Identifier*⟩ `end`
5	$\Rightarrow$ `def add(x,` ⟨*Identifier*⟩ ⟨*ParamTail*⟩`):` ⟨*Assignment*⟩ `return` ⟨*Identifier*⟩ `end`
6	$\overset{L}{\Rightarrow}^*$ `def add(x, y` ⟨*ParamTail*⟩`):` ⟨*Assignment*⟩ `return` ⟨*Identifier*⟩ `end`
7	$\Rightarrow$ `def add(x, y):` ⟨*Assignment*⟩ `return` ⟨*Identifier*⟩ `end`
8	$\Rightarrow$ `def add(x, y):` ⟨*Identifier*⟩ `=` ⟨*Expression*⟩ `return` ⟨*Identifier*⟩ `end`
9	$\overset{L}{\Rightarrow}^*$ `def add(x, y):  z =` ⟨*Expression*⟩ `return` ⟨*Identifier*⟩ `end`
10	$\Rightarrow$ `def add(x, y):  z =` ⟨*Operand*⟩ `+` ⟨*Operand*⟩ `return` ⟨*Identifier*⟩ `end`
11	$\Rightarrow$ `def add(x, y):  z =` ⟨*Identifier*⟩ `+` ⟨*Operand*⟩ `return` ⟨*Identifier*⟩ `end`
12	$\overset{L}{\Rightarrow}^*$ `def add(x, y):  z = x +` ⟨*Operand*⟩ `return` ⟨*Identifier*⟩ `end`
13	$\Rightarrow$ `def add(x, y):  z = x +` ⟨*Identifier*⟩ `return` ⟨*Identifier*⟩ `end`
14	$\overset{L}{\Rightarrow}^*$ `def add(x, y):  z = x + y return` ⟨*Identifier*⟩ `end`
15	$\overset{L}{\Rightarrow}^*$ `def add(x, y):  z = x + y return z end` $= f$

5.2.2 Leftmost expansion of ⟨*IfStmt*⟩

Table 3: Leftmost expansion of ⟨*IfStmt*⟩.

Stage	Sentential form
0	⟨*IfStmt*⟩
1	$\Rightarrow$ `if` ⟨*Condition*⟩`:` ⟨*Assignment*⟩ `else:` ⟨*Assignment*⟩ `end`
2	$\Rightarrow$ `if` ⟨*Operand*⟩ `>` ⟨*Operand*⟩`:` ⟨*Assignment*⟩ `else:` ⟨*Assignment*⟩ `end`
3	$\Rightarrow$ `if` ⟨*Number*⟩ `>` ⟨*Operand*⟩`:` ⟨*Assignment*⟩ `else:` ⟨*Assignment*⟩ `end`
4	$\overset{L}{\Rightarrow}^*$ `if 10 >` ⟨*Operand*⟩`:` ⟨*Assignment*⟩ `else:` ⟨*Assignment*⟩ `end`
5	$\Rightarrow$ `if 10 >` ⟨*Number*⟩`:` ⟨*Assignment*⟩ `else:` ⟨*Assignment*⟩ `end`
6	$\overset{L}{\Rightarrow}^*$ `if 10 > 5:` ⟨*Assignment*⟩ `else:` ⟨*Assignment*⟩ `end`
7	$\Rightarrow$ `if 10 > 5:` ⟨*Identifier*⟩ `=` ⟨*Expression*⟩ `else:` ⟨*Assignment*⟩ `end`
8	$\overset{L}{\Rightarrow}^*$ `if 10 > 5:  a =` ⟨*Expression*⟩ `else:` ⟨*Assignment*⟩ `end`
9	$\Rightarrow$ `if 10 > 5:  a =` ⟨*Operand*⟩ `else:` ⟨*Assignment*⟩ `end`
10	$\Rightarrow$ `if 10 > 5:  a =` ⟨*Number*⟩ `else:` ⟨*Assignment*⟩ `end`
11	$\overset{L}{\Rightarrow}^*$ `if 10 > 5:  a = 1 else:` ⟨*Assignment*⟩ `end`
12	$\Rightarrow$ `if 10 > 5:  a = 1 else:` ⟨*Identifier*⟩ `=` ⟨*Expression*⟩ `end`
13	$\overset{L}{\Rightarrow}^*$ `if 10 > 5:  a = 1 else:  a =` ⟨*Expression*⟩ `end`
14	$\Rightarrow$ `if 10 > 5:  a = 1 else:  a =` ⟨*Operand*⟩ `end`
15	$\Rightarrow$ `if 10 > 5:  a = 1 else:  a =` ⟨*Number*⟩ `end`
16	$\overset{L}{\Rightarrow}^*$ `if 10 > 5:  a = 1 else:  a = 2 end` $= i$

Thus

$$\langle Program \rangle \overset{L}{\Rightarrow}^* w,$$

where w is exactly the required program up to insignificant whitespace. Therefore $w \in L(G)$, which is the constructive membership proof requested by the assignment.

5.3 Parse Tree

A parse tree provides a structural representation of a derivation. The course notes state that, for a variable A, the existence of a derivation $A \Rightarrow^* \omega$ is equivalent to the existence of a parse tree with root A and yield ω [1]. Accordingly, Figure 1 shows the complete parse tree for the ⟨*IfStmt*⟩ constituent of the required program.

The full program tree additionally contains the top-level expansions

$$\langle Program \rangle \Rightarrow \langle StatementList \rangle \Rightarrow \langle Statement \rangle \, \langle StatementList \rangle,$$

where the first ⟨*Statement*⟩ expands to ⟨*FunctionDef*⟩ and the second expands to ⟨*IfStmt*⟩. The conditional subtree is shown in full because it illustrates the hierarchical structure while remaining readable. Every internal expansion in the tree corresponds to a production in Section 4.2 [3, 4].

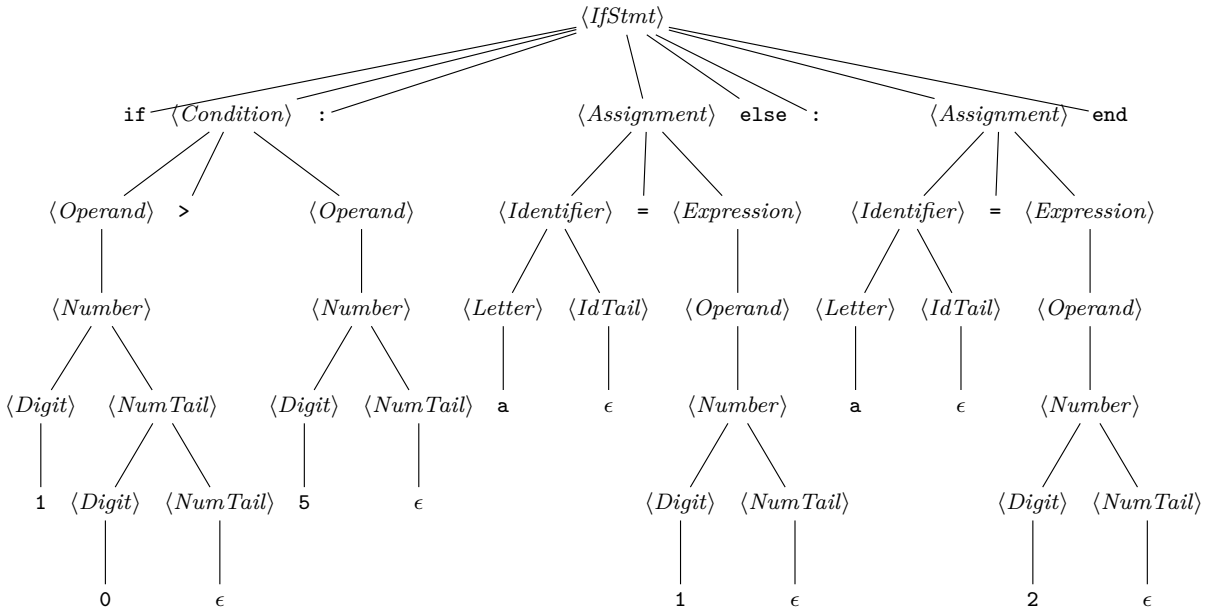

Figure 1: Complete parse tree for the ⟨*IfStmt*⟩ constituent of the required program.

Reading the terminal leaves of Figure 1 from left to right yields

`if 10 > 5:  a = 1 else:  a = 2 end.`

Thus the tree provides a structural representation of the same ⟨*IfStmt*⟩ derivation presented in Section 5.2.2.

6 Experimental Verification

6.1 Method

The derivation above serves as a constructive proof that the required string belongs to $L(G)$. As an additional empirical check, a minimal top-down recognizer was implemented. The recognizer implements the structural rules of the CFG together with the lexical convention that reserved keywords are distinguished from identifiers. For implementation efficiency, the lexical analysis phase converts valid identifiers to the `ID` token and decimal integers to the `NUM` token; this process implements the intended lexical behavior of the ⟨*Identifier*⟩ and ⟨*Number*⟩ components for the tested programs.

The validation process evaluates four test cases: the syntactically correct target program and three deliberately malformed variants. The recognizer is expected to strictly accept the valid program while accurately flagging the structural errors in the malformed inputs.

6.2 Results

The experiment confirms the intended behavior for the tested cases: the target program is recognized, while errors involving required delimiters are rejected. This experiment does not by itself prove every property of the full language; rather, it supports the design and independently checks the concrete membership result required by the assignment. A full proof that a grammar generates *exactly* a specified language would require proving both soundness (only intended

Test	Input modification	Result	Diagnostic
1	Required program exactly as specified	Accept	–
2	Remove the colon after `add(x, y)`	Reject	Expected `:`
3	Remove the colon after `else`	Reject	Expected `:`
4	Remove the final `end`	Reject	Expected `end`

Table 4: Experimental verification results.

strings are generated) and completeness (all intended strings are generated), as discussed by Rich [4]. For Assignment 1, the required constructive membership proof is supplied by the leftmost derivation.

7 Limitations

It is important to note that this CFG is intentionally designed as a minimal, pedagogical model rather than a comprehensive representation of Python's syntax. In particular:

- a function body contains exactly one assignment and one return statement;
- an `if-else` contains exactly one assignment per branch;
- only the comparison operator `>` is included;
- expressions are deliberately restricted to either a single operand or exactly one addition. A more comprehensive grammar would need additional nonterminals, such as $\langle \mathit{Term} \rangle$, $\langle \mathit{Factor} \rangle$, and $\langle \mathit{Unary} \rangle$, to encode operator precedence and associativity; this minimal design avoids those issues by limiting expression depth;
- indentation, type rules, name binding, scope, and semantic checks are outside the grammar; and
- `end` is used as an explicit block terminator because it appears in the problem statement.

These restrictions keep the grammar small enough that the required derivation can be inspected by hand. They can later be relaxed by adding recursive statement lists inside blocks, additional operators, more expression levels, loops, and other constructs.

8 Conclusion

A context-free grammar was designed for a small Python-like programming language. The grammar formally separates program structure, statements, function definitions, conditionals, assignments, expressions, conditions, identifiers, and numbers. A leftmost derivation shows constructively that the grammar generates

```
def add(x, y):  z = x + y return z end
  if 10 > 5:  a = 1 else:  a = 2 end.
```

A separate recognizer also accepts the required program and rejects three malformed variants.

A Verification Code

The following Python program was used for the experimental check. It is not required for the mathematical derivation, but it provides a reproducible test of the designed syntax.

```
import re

TOKEN_RE = re.compile(r"""
    (?P<WS>\s+)
  | (?P<NUM>\d+)
  | (?P<ID>[a-z][a-z0-9]*)
  | (?P<SYM>[(),:=+>])
""", re.VERBOSE)

KEYWORDS = {"def", "return", "end", "if", "else"}

def tokenize(text):
    pos, out = 0, []
    while pos < len(text):
        m = TOKEN_RE.match(text, pos)
        if not m:
            raise SyntaxError(f"Unexpected character at {pos}")
        kind, value = m.lastgroup, m.group()
        pos = m.end()
        if kind == "WS":
            continue
        if kind == "ID" and value in KEYWORDS:
            out.append((value, value))
        elif kind == "ID":
            out.append(("ID", value))
        elif kind == "NUM":
            out.append(("NUM", value))
        else:
            out.append((value, value))
    return out

class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.i = 0

    def peek(self):
        return self.tokens[self.i][0] if self.i < len(self.tokens) else "EOF"

    def eat(self, kind):
        if self.peek() != kind:
            got = self.tokens[self.i] if self.i < len(self.tokens) else ("EOF", "")
            raise SyntaxError(f"Expected {kind}, got {got}")
        self.i += 1

    def program(self):
        if self.peek() == "EOF":
            raise SyntaxError("Program cannot be empty")
        while self.peek() != "EOF":
            self.statement()

    def statement(self):
        if self.peek() == "def":
            self.function_def()
        elif self.peek() == "if":
            self.if_stmt()
        elif self.peek() == "ID":
```

```
            self.assignment()
        else:
            raise SyntaxError(f"Unexpected token {self.peek()}")

    def function_def(self):
        self.eat("def"); self.eat("ID"); self.eat("(")
        self.param_list()
        self.eat(")"); self.eat(":")
        self.assignment()
        self.eat("return"); self.eat("ID"); self.eat("end")

    def param_list(self):
        self.eat("ID")
        while self.peek() == ",":
            self.eat(","); self.eat("ID")

    def if_stmt(self):
        self.eat("if"); self.condition(); self.eat(":")
        self.assignment(); self.eat("else"); self.eat(":")
        self.assignment(); self.eat("end")

    def condition(self):
        self.operand(); self.eat(">"); self.operand()

    def assignment(self):
        self.eat("ID"); self.eat("="); self.expression()

    def expression(self):
        self.operand()
        if self.peek() == "+":
            self.eat("+"); self.operand()

    def operand(self):
        if self.peek() == "ID":
            self.eat("ID")
        elif self.peek() == "NUM":
            self.eat("NUM")
        else:
            raise SyntaxError("Expected ID or NUM")

def accepts(text):
    try:
        Parser(tokenize(text)).program()
        return True
    except SyntaxError:
        return False

target = """def add(x, y): z = x + y return z end
if 10 > 5: a = 1 else: a = 2 end"""

assert accepts(target)
assert not accepts(target.replace("add(x, y):", "add(x, y)", 1))
assert not accepts(target.replace("else:", "else", 1))
assert not accepts(target.rsplit(" end", 1)[0])
print("All four tests behaved as expected.")
```

References

[1] N. D. Han, "Topic 2: Context-free languages and pushdown automata," *F. Languages and the Theory of Computation*, Faculty of Mathematics and Informatics, Hanoi University of Science and Technology, course slides, 2026.

[2] M. Sipser, *Introduction to the Theory of Computation*, 3rd ed. Boston, MA, USA: Cengage Learning, 2013.

[3] P. Linz and S. H. Rodger, *An Introduction to Formal Languages and Automata*, 7th ed. Burlington, MA, USA: Jones & Bartlett Learning, 2023.

[4] E. Rich, *Automata, Computability and Complexity: Theory and Applications*. Pearson Education, 2007, rev. 2019.