CS3372 / Assignment 2

Abstract

This report extends the Python-like language from Assignment 1 and constructs a pushdown automaton (PDA) to accept it. The extended language adds nested blocks, while loops, arithmetic, comparison, and logical operators. A precedence-aware CFG defines the syntax, and a standard top-down CFG-to-PDA construction is used. A Python implementation simulates the PDA and is tested on valid and invalid programs, including the original Assignment 1 program.

1 Problem Statement

Assignment 2 has two objectives: extend the Python-like language from Assignment 1 with additional operations and control-flow statements, and design a Python implementation of a pushdown automaton that accepts the extended language.

The new language preserves the original syntax while adding logical and comparison operators, nested statements, and `while` loops. For example:

```
def choose(x, y):
    if x >= y and not x == 0:
        result = x
    else:
        result = y
    end
    return result
end

counter = 0
while counter < 3:
    counter = counter + 1
end
```

As in Assignment 1, whitespace and indentation are only for readability. Compound statements are explicitly closed by `end`.

2 Background

To define our programming language clearly, we use a context-free grammar (CFG). Formally, a CFG is written as a four-tuple:

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

where V is a finite set of variables, Σ is a finite terminal alphabet, R is a finite set of productions, and $S \in V$ is the start variable [1,2]. In simple terms, a CFG acts as a blueprint for the language, showing how expressions, loops, and nested blocks fit together [3].

To check whether a program follows these rules, we use a pushdown automaton (PDA). A PDA is essentially a standard finite-state machine with a memory stack attached [1, 4]. This stack gives the machine the memory it needs to handle nested structures, such as matching

parentheses or pairing each `while` loop with its closing `end` keyword [5]. We follow the standard seven-tuple definition with an explicit initial stack symbol Z_0:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F).$$

The components are the finite state set Q, input alphabet Σ, stack alphabet Γ, transition function δ, start state q_0, initial stack symbol Z_0, and accepting-state set F [4].

The core theoretical idea behind our design is that CFGs and PDAs have equivalent expressive power: any language defined by a CFG can be recognized by a PDA [1]. We use the standard top-down conversion method [5]. The PDA places the start variable on its stack, expands grammar variables using production rules, and matches literal tokens directly against the input code [5].

3 Design Progress

The extension was developed in the following stages:

1. **Preserve Assignment 1 constructs.** Function definitions, assignments, `if-else`, identifiers, numbers, and the original arithmetic/comparison forms remain valid.
2. **Generalize statement blocks.** Instead of allowing exactly one assignment in each branch, a block now contains one or more statements. Consequently, conditionals and loops may be nested.
3. **Add looping.** A `while` statement is introduced with the form `while Expression : StatementList end`.
4. **Extend arithmetic.** Addition is generalized to `+`, `-`, `*`, and `/`. Separate grammar levels are used so multiplication and division bind more tightly than addition and subtraction.
5. **Extend comparisons.** Six comparison operators are supported: `<`, `>`, `<=`, `>=`, `==`, and `!=`.
6. **Add logic.** The logical operators `and`, `or`, and `not` are introduced with explicit precedence levels.
7. **Convert the CFG to a PDA.** Rather than implementing only a stack check for block delimiters, the PDA simulates the complete grammar. It therefore validates the overall syntax, including operators, parentheses, statement structure, and nested blocks.
8. **Verify experimentally.** The Python program is tested on the Assignment 1 program, new valid programs, and deliberately malformed programs.

4 Extended Grammar Design

4.1 Language Specification

The language supports:

- one or more statements in a program or compound-statement block;
- function definitions with zero or more parameters;
- assignments and return statements;
- nested `if-else` statements;
- nested `while` loops;

- arithmetic expressions using `+`, `-`, `*`, and `/`;
- comparisons using `<`, `>`, `<=`, `>=`, `==`, and `!=`;
- logical expressions using `and`, `or`, and `not`;
- parenthesized expressions; and
- identifiers beginning with a lowercase letter, followed by lowercase letters, digits, or underscores, and nonnegative decimal integer literals.

The grammar deliberately describes a small Python-inspired language, not the complete Python grammar. In particular, explicit `end` tokens are retained from Assignment 1 to make nested block boundaries visible in the token stream.

4.2 Lexical Convention

Before the PDA parses any code, a small lexer turns raw source text into a clean stream of tokens. For example, any variable name (like `count` or `x`) becomes `ID`, and any number (like `42`) becomes `NUM`. Keywords (like `if`, `while`) and operators remain their own tokens.

We do this because a formal automaton needs a finite input alphabet Σ. If every possible variable name were treated as a unique symbol, the alphabet would be infinitely large [3].

The input alphabet is therefore:

$$\begin{aligned}\Sigma = \{&\texttt{ID}, \texttt{NUM}, \texttt{def}, \texttt{return}, \texttt{end}, \texttt{if}, \texttt{else}, \texttt{while},\\ &\texttt{and}, \texttt{or}, \texttt{not}, \texttt{(}, \texttt{)}, \texttt{,}, \texttt{:}, \texttt{=}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/},\\ &\texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{==}, \texttt{!=}\}.\end{aligned}$$

4.3 Formal CFG

Let

$$G = (V, \Sigma, R, S), \qquad S = \langle \mathit{Program} \rangle.$$

The variables are

$$\begin{aligned}V = \{&\langle \mathit{Program} \rangle, \langle \mathit{StatementList} \rangle, \langle \mathit{Statement} \rangle, \langle \mathit{FunctionDef} \rangle,\\ &\langle \mathit{ParamListOpt} \rangle, \langle \mathit{ParamList} \rangle, \langle \mathit{ParamTail} \rangle, \langle \mathit{IfStmt} \rangle, \langle \mathit{WhileStmt} \rangle,\\ &\langle \mathit{Assignment} \rangle, \langle \mathit{ReturnStmt} \rangle, \langle \mathit{Expression} \rangle, \langle \mathit{OrExpr} \rangle, \langle \mathit{OrTail} \rangle,\\ &\langle \mathit{AndExpr} \rangle, \langle \mathit{AndTail} \rangle, \langle \mathit{NotExpr} \rangle, \langle \mathit{Comparison} \rangle, \langle \mathit{ComparisonTail} \rangle,\\ &\langle \mathit{CompOp} \rangle, \langle \mathit{Arithmetic} \rangle, \langle \mathit{AddTail} \rangle, \langle \mathit{Term} \rangle, \langle \mathit{MulTail} \rangle, \langle \mathit{Factor} \rangle\}.\end{aligned}$$

The production set R is shown in Table 1.

Table 1: Production rules of the extended CFG.

Nonterminal		Production
$\langle \mathit{Program} \rangle$	$\rightarrow$	$\langle \mathit{StatementList} \rangle$
$\langle \mathit{StatementList} \rangle$	$\rightarrow$	$\langle \mathit{Statement} \rangle\ \langle \mathit{StatementList} \rangle \mid \langle \mathit{Statement} \rangle$
$\langle \mathit{Statement} \rangle$	$\rightarrow$	$\langle \mathit{FunctionDef} \rangle \mid \langle \mathit{IfStmt} \rangle \mid \langle \mathit{WhileStmt} \rangle \mid \langle \mathit{Assignment} \rangle \mid \langle \mathit{ReturnStmt} \rangle$
$\langle \mathit{FunctionDef} \rangle$	$\rightarrow$	`def ID (` $\langle \mathit{ParamListOpt} \rangle$ `) :` $\langle \mathit{StatementList} \rangle$ `end`
$\langle \mathit{ParamListOpt} \rangle$	$\rightarrow$	$\langle \mathit{ParamList} \rangle \mid \epsilon$
$\langle \mathit{ParamList} \rangle$	$\rightarrow$	`ID` $\langle \mathit{ParamTail} \rangle$
$\langle \mathit{ParamTail} \rangle$	$\rightarrow$	`,` `ID` $\langle \mathit{ParamTail} \rangle \mid \epsilon$
$\langle \mathit{IfStmt} \rangle$	$\rightarrow$	`if` $\langle \mathit{Expression} \rangle$ `:` $\langle \mathit{StatementList} \rangle$ `else :` $\langle \mathit{StatementList} \rangle$ `end`
$\langle \mathit{WhileStmt} \rangle$	$\rightarrow$	`while` $\langle \mathit{Expression} \rangle$ `:` $\langle \mathit{StatementList} \rangle$ `end`
$\langle \mathit{Assignment} \rangle$	$\rightarrow$	`ID =` $\langle \mathit{Expression} \rangle$

Nonterminal		Production
⟨*ReturnStmt*⟩	→	`return` ⟨*Expression*⟩
⟨*Expression*⟩	→	⟨*OrExpr*⟩
⟨*OrExpr*⟩	→	⟨*AndExpr*⟩ ⟨*OrTail*⟩
⟨*OrTail*⟩	→	`or` ⟨*AndExpr*⟩ ⟨*OrTail*⟩ | ϵ
⟨*AndExpr*⟩	→	⟨*NotExpr*⟩ ⟨*AndTail*⟩
⟨*AndTail*⟩	→	`and` ⟨*NotExpr*⟩ ⟨*AndTail*⟩ | ϵ
⟨*NotExpr*⟩	→	`not` ⟨*NotExpr*⟩ | ⟨*Comparison*⟩
⟨*Comparison*⟩	→	⟨*Arithmetic*⟩ ⟨*ComparisonTail*⟩
⟨*ComparisonTail*⟩	→	⟨*CompOp*⟩ ⟨*Arithmetic*⟩ | ϵ
⟨*CompOp*⟩	→	`<` | `>` | `<=` | `>=` | `==` | `!=`
⟨*Arithmetic*⟩	→	⟨*Term*⟩ ⟨*AddTail*⟩
⟨*AddTail*⟩	→	`+` ⟨*Term*⟩ ⟨*AddTail*⟩ | `-` ⟨*Term*⟩ ⟨*AddTail*⟩ | ϵ
⟨*Term*⟩	→	⟨*Factor*⟩ ⟨*MulTail*⟩
⟨*MulTail*⟩	→	`*` ⟨*Factor*⟩ ⟨*MulTail*⟩ | `/` ⟨*Factor*⟩ ⟨*MulTail*⟩ | ϵ
⟨*Factor*⟩	→	`ID` | `NUM` | `(` ⟨*Expression*⟩ `)`

Every production has a single variable on its left-hand side, so the grammar is context-free. The grammar is also written without left recursion. This form is convenient for top-down recognition because recursive extensions occur in tail variables such as ⟨*AddTail*⟩, ⟨*AndTail*⟩, and ⟨*StatementList*⟩.

4.4 Expression Precedence

Instead of throwing all operators into a single rule, our grammar splits expressions into several levels. This directly encodes the intended operator precedence. From highest to lowest priority, the order is:

$$\text{parentheses/factors} > \texttt{*}, \texttt{/} > \texttt{+}, \texttt{-} > \text{comparisons} > \texttt{not} > \texttt{and} > \texttt{or}.$$

For example, an expression like:

```
x + 1 * 2 >= y and not z == 0
```

is automatically grouped step-by-step as:

$$((x + (1 * 2)) \geq y)\ \texttt{and}\ \texttt{not}(z == 0).$$

The PDA does not compute the numerical result—it simply verifies that the tokens follow the correct structure defined by G.

4.5 Block Structure

The key upgrade from Assignment 1 is that compound blocks now hold a list of multiple statements (⟨*StatementList*⟩) rather than just a single line. This allows us to write nested code freely, such as:

```
while x < 10:
    if x != 3 or y >= 2:
        x = x + 1 * 2
    else:
        x = x + 2
    end
end
```

Each block explicitly closes with an `end` token. This makes it easy for the grammar to detect block boundaries without needing to track indentation.

5 PDA Design

5.1 Formal Definition

The PDA is the nondeterministic top-down automaton

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

where

$$\begin{aligned} Q &= \{q_0, q, q_f\}, \\ F &= \{q_f\}, \\ \Gamma &= V \cup \Sigma \cup \{Z_0\}. \end{aligned}$$

The transition function is

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*).$$

The input alphabet Σ is the token alphabet defined in Section 4.2. The stack alphabet contains all grammar variables and terminals because both kinds of symbols may appear in a sentential form stored on the stack.

The stack top is taken to be the leftmost symbol in the mathematical notation. The implementation stores the top at the right end of a Python tuple, so right-hand sides are pushed in reverse order.

5.2 Transition Construction

The PDA processes code using a straightforward "predict-and-match" strategy [5].

1. Initialization. Place the start variable $\langle \mathit{Program} \rangle$ onto the stack directly above Z_0:

$$\delta(q_0, \epsilon, Z_0) = \{(q, \langle \mathit{Program} \rangle Z_0)\}.$$

2. Expand a grammar variable. Whenever a grammar variable sits on top of the stack, replace it with the right-hand side of any valid production rule:

$$(q, \epsilon, A) \to (q, \alpha) \quad \text{for every } A \to \alpha \in R.$$

This step does not consume any input. It simply breaks down the high-level variable into smaller grammar components.

3. Match a terminal token. When a literal token reaches the top of the stack, check if it matches the current input token:

$$(q, a, a) \to (q, \epsilon).$$

If they match, consume the token from both the input stream and the stack.

4. Accept. Once the entire input has been read and only Z_0 remains on the stack:

$$\delta(q, \epsilon, Z_0) = \{(q_f, Z_0)\}.$$

The automaton moves to q_f, confirming the program is syntactically valid.

The standard top-down construction is nondeterministic because a nonterminal may have several possible productions. For example, $\langle \mathit{Statement} \rangle$ can expand to an assignment, a loop, or a conditional block. The PDA tries these possible paths; incorrect guesses hit a dead end and stop, while valid code finds at least one path that successfully reaches q_f [5, 6].

5.3 Why the PDA Accepts Exactly the Grammar Language

The connection between the grammar and the PDA is direct. If a program can be derived from the start variable:

$$\langle Program \rangle \Rightarrow^* w,$$

the PDA can recreate that exact derivation on its stack using ϵ-expansions. Whenever terminal tokens appear, the PDA matches and consumes them against the input code. This guarantees that every valid program has an accepting computation path in M.

Conversely, tokens can only be removed from the stack by matching real input, and variables can only change through valid rules in G. If code has a syntax error, it cannot complete a valid derivation and gets stuck. Formally, this top-down construction $NT(G)$ is proven to accept exactly $L(G)$ [1, 5].

5.4 Example PDA Computation

Consider the short program

```
x = 1 + 2
```

which the lexer converts to

`ID = NUM + NUM`.

A selected portion of one accepting computation is shown below. Only the relevant top of the stack is displayed.

Table 2: Selected steps of an accepting computation.

Unread input	Stack top	Action
`ID = NUM + NUM`	$\langle Program \rangle$	expand $\langle Program \rangle \to \langle StatementList \rangle$
`ID = NUM + NUM`	$\langle StatementList \rangle$	choose one $\langle Statement \rangle$
`ID = NUM + NUM`	$\langle Statement \rangle$	expand to $\langle Assignment \rangle$
`ID = NUM + NUM`	`ID`	match `ID`
`= NUM + NUM`	`=`	match `=`
`NUM + NUM`	$\langle Expression \rangle$	expand through expression levels
`NUM + NUM`	`NUM`	match first `NUM`
`+ NUM`	$\langle AddTail \rangle$	choose `+` $\langle Term \rangle$ $\langle AddTail \rangle$
`+ NUM`	`+`	match `+`
`NUM`	`NUM`	match second `NUM`
ϵ	tail variables	choose their ϵ productions
ϵ	Z_0	enter q_f

This example illustrates the two essential kinds of PDA moves: nondeterministic grammar expansion and deterministic terminal matching.

6 Python Implementation

6.1 Lexer

The lexer uses a regular expression to recognize whitespace, decimal integers, identifiers beginning with a lowercase letter and followed by lowercase letters, digits, or underscores, and punctuation/operators. Identifiers that equal a reserved keyword are kept as keyword tokens; other identifiers become `ID`; integers become `NUM`. Multi-character comparisons such as `>=` and `!=` are recognized before the one-character operators.

This preprocessing is necessary because a formal PDA operates over a finite alphabet. Treating every possible identifier spelling as a separate terminal would not produce a finite input alphabet.

6.2 Grammar Representation

The production relation is represented by a Python dictionary. Each key is a nonterminal and each value is a list of possible right-hand sides. A right-hand side is stored as a tuple, and the empty tuple represents ϵ.

For example:

```
"WhileStmt": [
    ("while", "Expression", ":", "StatementList", "end"),
],
"AddTail": [
    ("+", "Term", "AddTail"),
    ("-", "Term", "AddTail"),
    (),
],
```

6.3 Nondeterministic Search

In Python, each step of the PDA is represented by the tuple:

$$(\text{state}, \text{input position}, \text{stack}).$$

Because a variable can expand in multiple ways, the program uses a queue to explore all possible paths (breadth-first search). When the top of the stack is a variable, it branches out for every matching grammar rule. When it is a terminal, it only continues if the token matches the next input token.

To keep the search fast and prevent infinite loops, we apply two optimizations:

- A `visited` set tracks explored configurations so the program never explores the same configuration twice.
- A pruning check precalculates the minimum number of tokens each grammar symbol needs. If the symbols left on the stack require more tokens than what remains in the input, the program discards that branch immediately.

6.4 Acceptance Condition

The implementation accepts when the complete input has been consumed, the grammar symbols have been removed from the stack, only Z_0 remains, and the automaton enters q_f. Therefore, it checks more than balanced `end` markers: the complete token sequence must be generated by the CFG.

7 Experimental Verification

7.1 Test Method

Seven tests were used. The first verifies backward compatibility with the concrete Assignment 1 program. The next two exercise the added operators, loops, and nesting. The remaining tests deliberately violate the syntax.

All seven tests behaved as expected. In particular, the first result shows that the extension is backward compatible with the required Assignment 1 program, while the second and third results demonstrate the new language features.

Table 3: Experimental verification results.

No.	Test	Expected	Result
1	Assignment 1 program remains valid	Accept	Accept
2	Function with logical condition plus `while` loop	Accept	Accept
3	Nested `if` inside `while`	Accept	Accept
4	Missing final `end`	Reject	Reject
5	Malformed comparison operator `><`	Reject	Reject
6	Missing colon after `while` condition	Reject	Reject
7	Unbalanced parenthesis in expression	Reject	Reject

7.2 Interpretation of the Results

While seven test cases do not replace a full mathematical proof, they give practical confirmation that our implementation works properly. The successful runs on valid code show that the new loops, logical operators, and nested blocks work seamlessly alongside the original Assignment 1 syntax. The rejected cases provide additional evidence that the PDA checks the grammar structure rather than simply counting `end` markers.

8 Limitations

The language remains intentionally smaller than real Python. Important limitations are:

- indentation has no syntactic meaning; blocks use the explicit `end` token inherited from Assignment 1;
- `if` always contains an `else` branch;
- only `while` loops are included; `for`, `break`, and `continue` are not included;
- comparison chaining such as `x < y < z` is not supported;
- unary minus, exponentiation, modulo, strings, lists, function calls, and many other Python expressions are omitted;
- the grammar checks syntax only. Type checking, scope, declaration rules, return-position restrictions, division by zero, and other semantic properties are outside the PDA;
- a `return` statement is syntactically allowed anywhere that a statement is allowed, even though real Python applies additional contextual rules; and
- the general top-down NPDA is nondeterministic and can explore several grammar choices. The implementation uses memoization and safe minimum-yield pruning to keep the supplied examples practical.

These restrictions are deliberate. They keep the grammar understandable enough to inspect manually while still demonstrating the main Assignment 2 requirements: richer operations, conditionals, loops, nesting, and recognition by a pushdown automaton.

9 Conclusion

The Assignment 1 language was extended without discarding its original structure. The new grammar adds arithmetic, comparison, and logical operators, a `while` loop, general statement lists, nested blocks, return statements, and parenthesized expressions. Operator precedence is represented explicitly by separate grammar levels.

A nondeterministic top-down PDA was then constructed from the CFG. Its stack stores the current sentential form: nonterminals are expanded by ϵ-moves corresponding to productions, and terminal symbols are removed only when they match the next input token. Because this is the standard CFG-to-PDA construction, the design directly connects the grammar and the automaton instead of using the stack only for block-depth checking.

The Python implementation follows the same construction and successfully accepts the original Assignment 1 program and new programs containing logical expressions, comparisons, nested conditionals, and loops. It also rejects representative syntax errors. The resulting design therefore satisfies both the language-extension and PDA-implementation requirements of Assignment 2.

A Source Code

The complete Python implementation used for the experiments is reproduced below.

```
"""Nondeterministic PDA recognizer for the Assignment 2 language."""

from collections import deque
import re


GRAMMAR = {
    "Program": [("StatementList",)],
    "StatementList": [
        ("Statement", "StatementList"),
        ("Statement",),
    ],
    "Statement": [
        ("FunctionDef",),
        ("IfStmt",),
        ("WhileStmt",),
        ("Assignment",),
        ("ReturnStmt",),
    ],
    "FunctionDef": [
        (
            "def",
            "ID",
            "(",
            "ParamListOpt",
            ")",
            ":",
            "StatementList",
            "end",
        )
    ],
    "ParamListOpt": [("ParamList",), ()],
    "ParamList": [("ID", "ParamTail")],
    "ParamTail": [(",", "ID", "ParamTail"), ()],
    "IfStmt": [
        (
            "if",
            "Expression",
            ":",
            "StatementList",
            "else",
```

```
            ":",
            "StatementList",
            "end",
        )
    ],
    "WhileStmt": [
        ("while", "Expression", ":", "StatementList", "end")
    ],
    "Assignment": [("ID", "=", "Expression")],
    "ReturnStmt": [("return", "Expression")],
    "Expression": [("OrExpr",)],
    "OrExpr": [("AndExpr", "OrTail")],
    "OrTail": [("or", "AndExpr", "OrTail"), ()],
    "AndExpr": [("NotExpr", "AndTail")],
    "AndTail": [("and", "NotExpr", "AndTail"), ()],
    "NotExpr": [("not", "NotExpr"), ("Comparison",)],
    "Comparison": [("Arithmetic", "ComparisonTail")],
    "ComparisonTail": [("CompOp", "Arithmetic"), ()],
    "CompOp": [("<",), (">",), ("<=",), (">=",), ("==",), ("!=",)],
    "Arithmetic": [("Term", "AddTail")],
    "AddTail": [
        ("+", "Term", "AddTail"),
        ("-", "Term", "AddTail"),
        (),
    ],
    "Term": [("Factor", "MulTail")],
    "MulTail": [
        ("*", "Factor", "MulTail"),
        ("/", "Factor", "MulTail"),
        (),
    ],
    "Factor": [("ID",), ("NUM",), ("(", "Expression", ")")],
}

NONTERMINALS = frozenset(GRAMMAR)
KEYWORDS = frozenset(
    {"def", "return", "end", "if", "else", "while", "and", "or", "not"}
)

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


def tokenize(source):
    """Convert source text to the finite token alphabet used by the PDA."""
    tokens = []
    position = 0

    while position < len(source):
        match = TOKEN_RE.match(source, position)
        if match is None:
            fragment = source[position : position + 12]
            raise ValueError(
                "invalid character at position {}: {!r}".format(position, fragment)
```

```
            )

        kind = match.lastgroup
        value = match.group()
        position = match.end()

        if kind == "WS":
            continue
        if kind == "NUM":
            tokens.append("NUM")
        elif kind == "ID":
            tokens.append(value if value in KEYWORDS else "ID")
        else:
            tokens.append(value)

    return tokens


def compute_minimum_yields():
    """Find the minimum terminal length derivable from each nonterminal."""
    infinity = float("inf")
    minimum = {symbol: infinity for symbol in NONTERMINALS}

    changed = True
    while changed:
        changed = False
        for left, alternatives in GRAMMAR.items():
            for right in alternatives:
                length = 0
                for symbol in right:
                    if symbol in NONTERMINALS:
                        length += minimum[symbol]
                    else:
                        length += 1
                if length < minimum[left]:
                    minimum[left] = length
                    changed = True

    return minimum


MINIMUM_YIELD = compute_minimum_yields()


def minimum_stack_yield(stack):
    """Return the fewest input tokens that a stack can still generate."""
    length = 0
    for symbol in stack:
        if symbol == "Z0":
            continue
        if symbol in NONTERMINALS:
            length += MINIMUM_YIELD[symbol]
        else:
            length += 1
    return length


def accepts_tokens(tokens):
```

```
    """Simulate the top-down NPDA by breadth-first configuration search."""
    initial = ("q0", 0, ("Z0",))
    queue = deque([initial])
    visited = set()

    while queue:
        state, position, stack = queue.popleft()
        configuration = (state, position, stack)
        if configuration in visited:
            continue
        visited.add(configuration)

        if minimum_stack_yield(stack) > len(tokens) - position:
            continue

        if state == "q0":
            queue.append(("q", position, stack + ("Program",)))
            continue

        if state == "qf":
            if position == len(tokens):
                return True
            continue

        if stack == ("Z0",):
            if position == len(tokens):
                queue.append(("qf", position, stack))
            continue

        top = stack[-1]
        remainder = stack[:-1]

        if top in NONTERMINALS:
            for right in GRAMMAR[top]:
                # The tuple's right end is the stack top, so push in reverse.
                queue.append(("q", position, remainder + tuple(reversed(right))))
        elif position < len(tokens) and tokens[position] == top:
            queue.append(("q", position + 1, remainder))

    return False


def accepts(source):
    """Return True exactly when source is a syntactically valid program."""
    try:
        return accepts_tokens(tokenize(source))
    except ValueError:
        return False


TESTS = [
    (
        "Assignment 1 program",
        """
        def add(x, y):
            z = x + y
            return z
        end
```

```
        if 10 > 5:
            a = 1
        else:
            a = 2
        end
        """,
        True,
    ),
    (
        "logical condition and while loop",
        """
        def choose(x, y):
            if x >= y and not x == 0:
                result = x
            else:
                result = y
            end
            return result
        end
        counter = 0
        while counter < 3:
            counter = counter + 1
        end
        """,
        True,
    ),
    (
        "nested if inside while",
        """
        while x < 10:
            if x != 3 or y >= 2:
                x = x + 1 * 2
            else:
                x = x + 2
            end
        end
        """,
        True,
    ),
    ("missing final end", "while x < 3: x = x + 1", False),
    ("malformed comparison", "if x >< 3: x = 1 else: x = 2 end", False),
    ("missing while colon", "while x < 3 x = x + 1 end", False),
    ("unbalanced parenthesis", "x = (1 + 2", False),
]


def run_tests():
    for number, (name, program, expected) in enumerate(TESTS, start=1):
        result = accepts(program)
        status = "PASS" if result == expected else "FAIL"
        decision = "Accept" if result else "Reject"
        print("{}: {} - {} ({})".format(number, status, name, decision))


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

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

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

[5] J. C. Martin, *Introduction to Languages and the Theory of Computation*, 4th ed. New York, NY, USA: McGraw-Hill, 2011.

[6] D. C. Kozen, *Automata and Computability*. New York, NY, USA: Springer, 1997.