🍳 Cook++ Programming Language for Cooking Recipes

Powered by *PAN Analyzer*

A syntax analyzer that parses and validates Cook++ programs.

Group Members:

Bernabe, Jan Allen

Labasan, Jessa Mae

Tiu, Christian Harold

Programming Languages (CCPGLANG / COM223)

College of Computing and Information Technologies

National University – Manila

Instructor: Ms. Armida P. Salazar

Date of Submission: October 21, 2025

1. Purpose

The **Cook++ Programming Language** is a simple, domain-specific language designed to represent cooking instructions in a structured, programmable format. It bridges the gap between everyday kitchen activities and fundamental programming concepts, making it an ideal educational tool for beginners learning the logic behind code execution.

Goals

- **Educational Focus** – Teach core programming constructs (variables, conditionals, loops, and functions) through familiar cooking metaphors.
- **Intuitive Syntax** – Use natural, cooking-related keywords that are immediately understandable to beginners.
- **Error Prevention** – Provide clear, contextual error messages that help users learn from mistakes.
- **Real-World Mapping** – Demonstrate how programming abstractions can model everyday tasks like cooking.
- **Structured Recipes** – Enable systematic organization of cooking instructions that can be reused and modified like real programs.

Target Users

Primary Audience:

- High school or college students taking introductory programming courses.
- Self-taught beginners seeking a fun and intuitive learning approach.
- Educators looking for engaging teaching materials to explain syntax and logic.

Secondary Audience:

- Home cooks are interested in experimenting with programmable recipes.
- Culinary students exploring computational thinking in kitchen workflows.

Use Cases

- **Educational Settings** – Teaching sequence, selection, and iteration using real-life analogies.
- **Recipe Management** – Writing reusable, parameterized “recipe functions.”
- **Algorithm Visualization** – Demonstrating structured control flow through a relatable theme.

2. Design Criteria and Justification

Readability

Cook++ prioritizes readability by adopting natural-language keywords such as mix, chop, and boil that mirror real cooking verbs. Minimal punctuation (limited to {}, (), and ;) ensures that even non-programmers can intuitively follow program flow.

Example comparison:

```
// Traditional syntax

for (int i = 0; i < 3; i++) {

    stir();

}

// Cook++ syntax

repeat 3 times {

    stir;

}
```

Justification: This natural syntax reduces cognitive load and helps beginners focus on logic rather than syntax rules. Studies in computer science education support natural-language approaches for improving comprehension.

Writability

Cook++ is designed to be concise and expressive:

- **Single-word commands** like fry, mix, and serve.
- **Flexible expressions** supporting mathematical and conditional logic.
- **Simple procedure definitions** using procedure and return.

Justification: Good writability encourages experimentation and creativity. Learners can quickly modify or combine commands to explore different outcomes.

Reliability

By enforcing structured grammar and block syntax, **Cook++** prevents many common beginner mistakes such as missing braces or misaligned statements. The **PAN Analyzer** verifies each token and ensures strict adherence to the language's grammar.

Cost

Cook++ and the PAN Analyzer are implemented entirely in **Python**, making them lightweight, portable, and free to use. No external dependencies are required beyond standard Python libraries.

3. Overview of Language Features

Cook++ introduces a set of intuitive, recipe-inspired commands that directly map to fundamental programming concepts. The language supports a variety of features that make it both simple and expressive for beginners. At its core, Cook++ allows users to define procedures, perform assignments, use control structures, and call reusable instructions—all while maintaining the readability of a cooking recipe.

Core Commands

Cook++ includes core commands such as add, mix, chop, boil, fry, bake, and serve, which act as built-in functions representing real-world cooking actions. These statements always end with a semicolon to signify the completion of a step. By structuring instructions in this way, the language reads naturally, like a step-by-step recipe, while still following strict programming syntax rules.

Variables and Assignment

Variables in Cook++ are declared and assigned values using the set keyword. For instance, set ingredients = 5; creates a variable named ingredients and assigns it the value 5. This mirrors the process of setting ingredients before cooking, helping beginners understand variable initialization and assignment in a familiar context.

Conditionals

Cook++ supports conditional statements using if and else blocks. Conditions are enclosed in parentheses and followed by braces that define the statement blocks. This structure allows for branching logic similar to real-world cooking decisions, such as:

```
if (temperature > 100) {

   boil water;

} else {

   wait;

}
```

This feature introduces users to logical reasoning, comparison operators, and control flow.

Loops

The language provides looping constructs through the repeat n times syntax. This construct allows a block of instructions to run multiple times, making it ideal for repetitive actions like stirring or mixing:

```
repeat 3 times {

    stir;

}
```

This loop mechanism is simpler and more readable than traditional for loops, reducing syntactic complexity while preserving the underlying logic of iteration.

Procedures and Function Calls

Cook++ enables modular and reusable code through the procedure keyword. A procedure defines a sequence of actions that can later be invoked using its name and parameters. For example:

```
procedure lutongUlam(sangkap) {

    chop sangkap;

    fry sangkap 10;

    mix;

    return;

}
```

Users can then call it simply by writing lutongUlam(gulay);, demonstrating the principles of abstraction and encapsulation in a relatable form.

Expressions and Arithmetic

Cook++ supports arithmetic operations such as addition, subtraction, multiplication, and division using standard operators (+, -, *, /). These can be used inside expressions or assignments, e.g., set total = sugar + flour;. This feature introduces basic computation and expression parsing.

Error Handling

One of the core goals of the PAN Analyzer is to provide informative and beginner-friendly feedback. When syntax errors occur—such as missing semicolons, unmatched braces, or incorrect keywords—the analyzer displays precise error messages that indicate the line and column where the issue occurred. This feature helps learners quickly identify and correct mistakes, reinforcing proper coding habits.

4. BNF Grammar (12 Productions)

```
<program> ::= { <statement> }+

<statement> ::= <procedure_decl> | <assignment> | <if_stmt> | <loop_stmt> | <action> |
<function_call> | <return_stmt> | <print_stmt>

<procedure_decl> ::= "procedure" <identifier> "(" [ <parameters> ] ")" "{" { <statement> } "}"

<assignment> ::= "set" <identifier> "=" <expression> ";"

<if_stmt> ::= "if" <condition> "{" { <statement> } "}" [ "else" "{" { <statement> } "}" ]

<loop_stmt> ::= "repeat" <number> "times" "{" { <statement> } "}"

<action> ::= ("chop" | "fry" | "boil" | "mix" | "stir" | "bake" | "wait" | "serve") [ <identifier> ] [
<number> ] ";"

<function_call> ::= <identifier> "(" [ <arguments> ] ")" ";"

<return_stmt> ::= "return" ";"

<print_stmt> ::= "print" <string> ";"

<expression> ::= <term> { ("+" | "-" | "*" | "/") <term> }

<term> ::= <identifier> | <number>

<condition> ::= <expression> <comparison_op> <expression>

<comparison_op> ::= "==" | "!=" | "<" | ">" | "<=" | ">="
```

Covers all major constructs: declarations, assignments, conditionals, loops, function calls.

5. Syntax Analyzer Implementation (PAN Analyzer)

Tool Name: PAN Analyzer
Language Used: Python
Parser Type: Recursive Descent Parser

Description

The **PAN Analyzer** is the core syntax analysis tool for Cook++. It takes a Cook++ source file as input, tokenizes it, validates the syntax based on the BNF grammar, and outputs either:

- A **success message** ("No syntax errors found"), or
- A **detailed error message** indicating the line and type of syntax error.

Implementation Components

1. **Lexer:** Converts raw code into tokens (keywords, identifiers, numbers, symbols).
2. **Parser:** Checks token order and structure according to grammar rules.
3. **Error Reporter:** Displays informative messages for debugging syntax errors.

6. Test Coverage (Sample Programs)

Program 1: Basic Actions

```
chop gulay;

fry gulay 10;

mix;

serve;
```

Program 2: Procedure and Call

```
procedure lutongUlam(sangkap) {

    chop sangkap;

    fry sangkap 10;

    mix;

    return;

}

lutongUlam(gulay);

serve;
```

Program 3: Conditional

```
if gulay == fresh {

    fry gulay 5;

} else {

    boil gulay;

}

serve;
```

Program 4: Loop

```
repeat 3 times {

    mix;

    wait 1;

}
```

```
serve;
```

Program 5: Combination

```
set count = 2;

procedure luto(sangkap) {

    repeat count times {

        chop sangkap;

        fry sangkap 5;

    }

    serve;

}

if count > 1 {

    luto(gulay);

} else {

    boil gulay;

}
```

7. Conclusion

The **Cook++ Language** and its **PAN Analyzer** successfully demonstrate how programming language design principles can be introduced through a creative and relatable concept: cooking. By combining intuitive syntax, structured grammar, and real programming constructs, the project achieves both educational and technical goals — serving as a fun yet powerful model for learning programming fundamentals.