A-Level Computer Science Project
Cambridge Pseudocode → Python Translator (Partial Compiler)

Project Description

1 Project Goal

Your task in this project is to design and implement a **partial compiler** that translates a subset of **Cambridge International pseudocode** into valid **Python 3 programs**.

Your program will:

1. Ask the user for the name of a text file containing pseudocode.
2. Read the file, analyse it, and convert it into an internal structure.
3. Produce:
    - a generated Python program saved in a separate `.py` file, and
    - additional intermediate "transition files" that show the internal representations used at different stages of the translation. These files are for learning purposes and will help you reflect on how compilers process source code.
4. Report basic errors (e.g., file not found, unsupported construct, invalid syntax).

This project is intended to help you deepen your understanding of:

- how compilers work,
- the stages involved in translation,
- the relationship between syntax and meaning,
- and the connection between Cambridge-style pseudocode and real programming languages.

2 Supported Pseudocode Subset

Your translator is **not** expected to support the entire CIE pseudocode language. Instead, it must correctly handle the specific subset listed below.

You must support:

2.1 Comments

You should filter out single-line comments starting with `//`, whether at the start of a line or after code. They should not appear in the generated Python code.

2.2 White space

We are going to consider that every word and symbol in the input file should be separated by at least one space.

2.3 Data and Variables

- integer, real, boolean, and string literals
- variable names using letters, digits, and underscores
- assignment using the CIE operator: `<-`
- declaration of variables using: `DECLARE x : INTEGER`

2.4 Input and Output

```
INPUT x
OUTPUT expression
```

`INPUT` should read a value either as string, int, or float based on context

2.5 Flow Control

IF statements:

```
IF condition THEN
    statements
ELSE
    statements
ENDIF
```

WHILE loops:

```
WHILE condition DO
    statements
ENDWHILE
```

2.6 operators

Expressions are going to be limited to single operations using the following operators: For that reason, parentheses are not required or supported.

Arithmetic:

```
+   -   *   /
```

Comparison:

```
=   <>   <   >   <=   >=
```

Boolean:

```
AND   OR   NOT
```

2.7 Not Included

The following features are *not* part of this project:

- arrays
- for loops
- switch/case statements
- multiple operators in a single expression, parentheses.
- exponentiation

- functions or procedures
- user-defined subroutines
- built-in string manipulation functions
- file handling
- records / classes / objects

Your translator only needs to work correctly for the subset above.

3 Lexical analysis

3.1 Preparations

Draw out the steps you will need to take during lexical analysis.

1. _____ : _____
2. _____ : _____
3. _____ : _____

Then create your work folder, and create a new Python file called `lexer.py`.

3.2 _____

In this step, we will write a helper functions and classes for us to use in the next step. Here are their signatures, fill in their description:

Helper class:

```
class CleanLine:
    """Class representing a cleaned line of code."""
    def __init__(self, content : str, line_number :
       int):



    # potentially add __str__ or __repr__ methods
       after
```

And helper functions:

```
def is_blank_line(line : str) -> bool:
    # Write your description of this function here
```

```
10 def remove_comments_from_line(line: str) -> str:
11     # Write your description of this function here
12
13
14
15
16
17 def create_cleaned_lines(raw_lines: list[str]) ->
     list[CleanLine]:
18     # Write your description of this function here
19
20
21
22
23
24
25
26     # end of create_cleaned_lines
```

3.3 _____

Using the list of supported features, write down a list of all tokens you will need to identify during lexical analysis.
Each specific keyword, operator, or symbol should have its own token.
Identifiers and literals can be grouped into broader categories, depending on your design.

In lexer.py, let's first create a class to represent tokens:

```
class Token:

    """Class representing a token."""
    def __init__(self, type: str, value: str,
       line_number: int):
    # fill in the constructor method


    def __repr__(self):
    # fill in the representation method
```

In order to separate the cleaned lines into tokens, we will split each line into words based on spaces. Note that word and token are not equivalent. For example, a string literal like `"Hello, World!"` is a single token but contains spaces.

We want to write a function with the following signature to perform lexical analysis:

```
def lexical_analysis(cleaned_lines: list[clean_line
   ]) -> list[Token]:
```

In order to do that, we will need to implement these intermediates:

```
def is_first_keyword_or_identifier(words: list[str])
    -> bool:
    # Write your description of this function here


def is_first_number_literal(words: list[str]) ->
   bool:
    # Write your description of this function here


def is_first_string_literal(words: list[str]) -> int
   :
    # Write your description of this function here


```

```
13 def is_first_operator(words: list[str]) -> bool:
14     # Write your description of this function here
15
16
17 def tokenize_line(words: list[str], line_number: int
   ) -> list[Token]:
18     # Write your description of this function here
```

Now, we implement the `lexical_analysis` function using the helper functions above.

If should check the first word of our list of words, and depending on what it is, create the appropriate token, remove the relevant words from the list, and continue until there are no words left.

Take some time (and writing space) to plan the structure of this function before you start coding:

Once this is done and implemented, you can test your lexer on some sample pseudocode files to see if it correctly identifies and outputs the tokens. Remember that you should output the tokens to a text file for later inspection.

4 Parsing / Syntax Analysis

Parsing is about understanding the structure of pieces of code based on the rules of the language. We use the tokens generated by the lexer, and a set of rules called a **grammar** to build abstract representations of the code called **abstract syntax trees** (ASTs). For example, a complex expression like:

```
IF x < 10 AND y >= 5 THEN
    OUTPUT "In range"
ELSE
    OUTPUT "Out of range"
ENDIF
```

can be abstracted into a data type that represents the IF statement, and stores:

- the condition (which itself is an expression tree),
- the list of statements to execute if true,
- and the list of statements to execute if false.
- the line number where the IF statement begins.

This can then be used in later stages of the compiler to generate code in the target language (Python), or any other language for that matter.

4.1 Preparations

Using the supported pseudocode subset, write out the grammar rules that define the structure of valid pseudocode programs in this subset. Use BNF notation. and don't hesitate to define helper rules to keep things clear.

- $< litteral >::=$

- $< identifier >::=$

- $< unary\_expression >::=$

- $< binary\_expression >::=$

- $< statement\_list >::=$

- $< assignment\_statement >::=$

- $< declare\_statement >::=$

- $< input\_statement >::=$

- $< output\_statement >::=$

- $< if\_statement >::=$

- $< while\_statement >::=$

- $< statement >::=$

4.2 Making AST Node Classes

Based on your grammar, create Python classes to represent the different types of AST nodes you will need. Write them in a new file called `AST.py`.

Then should all derive from a common base class called `ASTNode`:

```
class ASTNode:
    """ Base class for all AST nodes. """
    pass
```

For example, here is how you might define an AST node for an assignment statement:

```
class AssignmentNode(ASTNode):
    """ AST node representing an assignment
       statement. """
    def __init__(self, variable_name: str,
       expression: ASTNode, line_number: int):
        self.variable_name = variable_name
        self.expression = expression
        self.line_number = line_number
```

Continue defining classes for all other constructs in your grammar, such as literals, expressions, input/output statements, control flow statements, etc. (everything you just wrote the BNF for).

4.3 Implementing the Parser

Create a new file called `parser.py`.

In this file, we are going to implement what's called **recursive descent parsing**.

A parser function is a function that looks at the flow of tokens and tries to match them against the grammar rules you defined earlier. For that, it needs to be able to look at the current token, and move forward in the list of tokens as it matches them.

So, we need some helper functions first:

Here, we will say we "consume" a token when we remove it from the list of tokens.

```
    ### Helper functions for parsing tasks ###

def peek_token(tokens : list[Token]):
    """Return next token without consuming it."""
    # implementation here

def advance_token(tokens : list[Token]):
    """Consume and return next token."""
    # implementation here

def expect_token(tokens : list[Token],
   expected_types : list[str]):
    """Consume the next token if it matches the
       expected types, else raise an error."""
    # implementation here
```

Now, we can start implementing parser functions for each grammar rule you defined earlier. Each function should return an AST node representing the construct it parsed. These functions will call each other recursively to build the full AST.

Here's an example of how you might implement a parser function for an assignment statement:

```
def parse_assignment_statement(tokens : list[Token])
    -> AssignmentNode:
    """Parse an assignment statement and return an
       AssignmentNode."""

    # assignments start with an identifier
    identifier_token = expect_token(tokens, ["
       IDENTIFIER"])
```

```
6
7       # then expect the assignment operator, if
            something else is found, expect_token will
            raise an error
8       expect_token(tokens, ["ASSIGN"])
9
10      # then parse the expression on the right side
11      expression_node = parse_expression(tokens)
12
13      # return the constructed AssignmentNode, note
            that the "assign" token was consumed but isn'
            t needed in the node
14      return AssignmentNode(identifier_token.value,
            expression_node, identifier_token.line_number
            )
```

Continue implementing parser functions for all other constructs in your grammar, such as literals, expressions, input/output statements, control flow statements, etc.

Finally, implement a top-level function that starts the parsing process and returns the root of the AST for the entire program. By the way, a program is just a list of statements...

4.4 Printing the AST

To help with debugging and understanding the structure of the AST, implement a function to print the AST in a readable format. You should add a `print_tree(indentation = "")` method to each AST node class to facilitate this.

for example, in the `AssignmentNode` class, you might add:

```
1       def print_tree(self, indentation = ""):
2           print(f"{indentation}AssignmentNode:")
3           print(f"{indentation}    {self.variable_name
                }")
4           self.expression.print_tree(indentation + "
                ")
```

Then, in your main parser file, you can implement a function to print the entire AST starting from the root node. Don't overthink it, the recursive calls will take care of everything.

5 Code Generation

The final stage of your compiler is to generate Python code from the AST produced by the parser.

Once again, we will implement this in a new file called `codegen.py`. Although the heavy lifting will be done by methods in the AST node classes.

In each AST node class, implement a method called `generate_python(indentation = "")` that returns a string containing the Python code for that node.

For example, in the `AssignmentNode` class, you might add:

```
1    def generate_python(self, indentation = "") ->
        str:
2        expr_code = self.expression.generate_python
            ()
3        return f"{indentation}{self.variable_name} =
             {expr_code}\n"
```

Some CIE pseudocode constructs will require more complex translations. For example, CIE INPUT statements need to determine the expected data type based on context, which may require additional logic in the code generation method.

Continue implementing the `generate_python` method for all other AST node classes, ensuring that each method correctly translates the pseudocode construct into Python syntax.

Finally, implement a top-level function in `codegen.py` that takes the root of the AST and generates the complete Python program as a string. This function should call the `generate_python` method on the root node.

6 Testing

Thoroughly test your compiler with various pseudocode programs that cover all supported constructs. Ensure that the generated Python code is correct and behaves as expected.

You should have learned everything about testing strategies last year, so apply those principles here. I want you to test normal, edge cases, and also error cases.