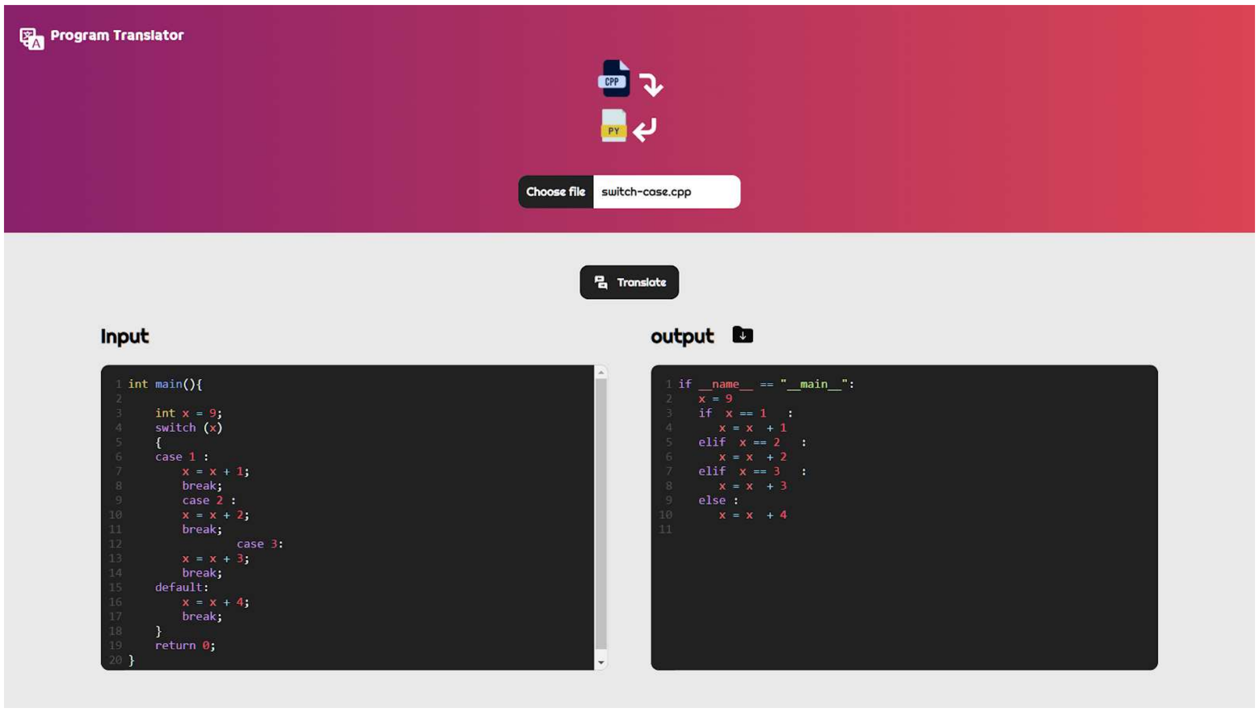
Compiler Design

Program Translator

From C++ to PYTHON



Input

```
int main(){

    int x = 9;
    switch (x)
    {
    case 1 :
        x = x + 1;
        break;
        case 2 :
        x = x + 2;
        break;
                case 3:
        x = x + 3;
        break;
    default:
        x = x + 4;
        break;
    }
    return 0;
}
```

output

```
if __name__ == "__main__":
   x = 9
   if  x == 1   :
      x = x  + 1
   elif  x == 2   :
      x = x  + 2
   elif  x == 3   :
      x = x  + 3
   else :
      x = x  + 4
```

Team Names:

Ziad Asem Mohamed

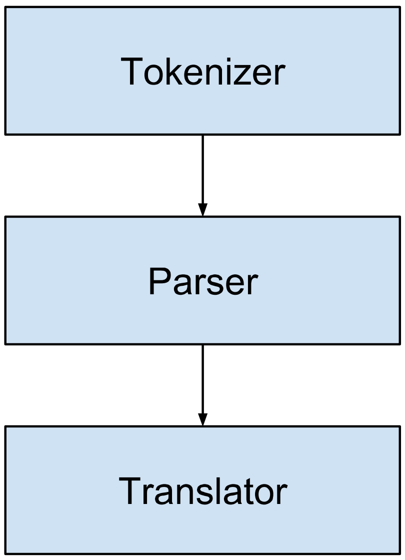
Ahmed Osman Ali

Ramy Mohamed Sayed Abdel-Khalik

Project Introduction:

Develop a program that automates translating C++ code into Python code. The program should take as input a C++ source file and generate an equivalent Python source file, preserving the functionality and structure of the original code. The goal is to provide a seamless and accurate conversion from C++ to Python, minimizing the need for manual code rewriting and ensuring that the resulting Python code remains readable and maintainable.

The program should handle a wide range of C++ language features, including but not limited to variable declarations, control flow statements (such as if-else, loops), functions, classes, and standard library functions. It should consider different data types, operator overloading, and C++-specific features such as pointers and references, and aim to convert them into their Python equivalents or appropriate alternatives.



The translation process should adhere to Python's syntax and coding conventions, striving for idiomatic Python code. It should address potential challenges, such as differences in memory management, exception handling, and input/output mechanisms between C++

and Python. The program should also provide informative error messages and gracefully handle any unexpected or unsupported C++ constructs.

The translated Python code should be functionally equivalent to the original C++ code, producing the same output or behavior when executed. The program should strive for accuracy, efficiency, and maintainability in the conversion process, ensuring that the resulting Python code is readable, understandable, and easily modifiable.

However, for now the program just translates the basic statements (if-else, while, do-while, switch etc.)

Tokenizer:

The tokenizer takes input from the C++ program to start tokenizing it , In compiler design, the tokenizer, also known as the lexer or scanner, breaks down the source code into meaningful tokens or lexemes. It applies predefined rules to identify keywords, identifiers, operators, literals, and punctuation symbols. The tokenizer removes unnecessary characters like white spaces and comments, generating a stream of tokens for further compilation phases. Its main purpose is to provide a structured representation of the code, simplifying subsequent analysis and processing by the compiler. Ultimately, the tokenizer enables accurate translation of the code into executable form by transforming the source code into a sequence of meaningful tokens.

Parser:

The parser in compiler design is responsible for analyzing the sequence of tokens generated by the tokenizer and ensuring that it follows the grammar rules of the programming language. It constructs a hierarchical structure, such as an abstract syntax tree (AST), to represent the syntactic structure of the code. The parser uses grammar and parsing algorithms to match tokens against the grammar rules, grouping them into higher-level language constructs. It may also perform semantic analysis to enforce correctness and consistency. The parser's main purpose is to validate the code's syntax, provide a structural representation, and enable further analysis and transformation in the compilation process. Ultimately, it plays a crucial role in converting the source code into executable form.

CFG:

main_stmt -> int main ( ) { stmts }

stmts-> stmt stmts | e

Stmt -> asgmt_stmt| if_stmt | while_stmt |switch_case | do_while | for_stmt

Asgmt_stmt -> id = expr ;

if_stmt -> if (cond) { stmts } elifs els

elifs -> elif elifs | e

elif -> elif {stmts}

els -> else {stmts}

while_stmt -> while(conds){stmts}

do_while -> do{stmts}while(conds)

For_stmt -> for(asmgt conds ;id ++){stmts}

Switch_case -> switch(id){ switch_body}

Switch_body -> switch_line Switch_body | default_line | e

switch_line -> case digit : stmts break ;

Conds -> cond logicOpr conds | e

logicOpr -> && | || | e

Cond -> id relop digit

expr → term rest

rest → + term rest | - term rest | ε

term → factor rest1

rest1 → * factor rest1 | / factor rest1 | ε

factor → digit | (expr)

digit → 0 | 1 | … | 9

Id -> a|…|z|A|….|Z

Translator:

The translator takes input from the output of the parser as a parse tree and starts translating the C++ statements into the equivalent statement in python, taking of interoperability and the translation of statements that might not be supported in Python, such as switch case or the increment operator, etc.

A C++ to Python-translator is designed to convert C++ source code into an equivalent representation in the Python programming language. It performs various stages, including lexical analysis, parsing, semantic analysis, optimization, and code generation, to ensure the accurate conversion of C++ code to Python.

The translator analyzes the C++ code's syntax and semantics, validating its correctness and compliance with Python's rules. It resolves identifiers, handles data type differences, and converts C++-specific constructs to their Python counterparts. The translator also performs optimizations to improve the efficiency and performance of the resulting Python code.

Once the analysis and optimization stages are completed, the translator generates Python code that can be executed directly in a Python environment. The generated Python code maintains the functionality and behavior of the original C++ code.

The primary purpose of a C++ to Python-translator is to enable the migration of C++ codebases to Python, automating the conversion process and saving developers from manually rewriting the code. It simplifies the transition between the two languages, allowing developers to leverage the benefits of Python while preserving the existing functionality of their C++ code.

In summary, a C++ to Python-translator serves the purpose of converting C++ code to an equivalent Python representation, facilitating code migration, maintaining functionality, and providing developers with an automated solution for translating C++ projects to Python.

Project strengths :

- Translate statements in C++ that don't exist in Python (e.g. do-while statements and switch case statements)
- Detect syntax errors in source code
- Detect relations between complex statements (e.g. for statement inside while statement inside if statement,.etc)
- Detect relations between compound conditions (e.g. A&&B || C)
- Responsive and user-friendly UI
- Portable

Project limitations:

- The translator can't detect semantic errors in the source code
- Translate for-statement with a single step and single increment variable