

CS 4308 – Concepts of Programming Languages

Course Project Deliverables

To complete the CS4308 project, a language processor (interpreter) must be developed for a specific subset of the SCL language. The interpreter can be written in any of the following programming languages: C, C++, C#, Java, Python, or Ada.

The language processor should be capable of processing an SCL program, with each token in the language being separated by white spaces. The parsing algorithm must be able to detect any syntactical errors and print an appropriate error message for each error found. In addition, any run-time errors must also be detected, and appropriate error messages should be printed.

To fulfill the requirements of this project, a complete software development process must be applied. This includes ensuring that the source code (of the scanner, parser, and complete interpreter) is well-structured and easy to understand. The code should also include comments to help clarify the implementation.

It is important not to hard-code input data in the source code. Instead, appropriate input statements should be used to ensure that the purpose of software development is not defeated.

Please do not include any compiled files and/or IDE project files in your submission only. Your complete submission should consist of a well-written report (refer to 'submission report.pdf'), the subset of the SCL grammar used (in BNF or EBNF notation), source code files for the implementation, and the input SCL program file used to test your project. All of these files should be submitted as a single archive.

Your report should document the work done, including an explanation of how to run your program and the input and output produced when running your program.

Deliverables (see course *Module's schedule for due dates*):

1. Module 3 – 1st Deliverable

The task at hand is to develop an interpreter for a subset of the SCL language. SCL is an experimental system programming language that can be accessed at <http://ksuweb.kennesaw.edu/~jgarrido/sysplm/>. To accomplish this, you must implement a scanner that reads the source code and creates a list of tokens. The scanner implementation must include an array of the keywords used in the subset of SCL, an array (or list) of the identifiers (variables), and other tokens (such as operators, keywords, constants, and/or special characters). Additionally, you will need to define the grammar of the subset of SCL that you are using in BNF/EBNF.

Python is recommended as the programming language of choice for this project. The program should take command-line input for a file and generate a JSON file with a list of tokens while also printing those tokens to the console. For instance, the following command will execute the program on the given source SCL file: **python scl_scanner.py hello_world.scl**

You must submit a brief report detailing the work done, including the grammar of the subset of SCL, source code files for the scanner program, as well as input and output files. The report should demonstrate the execution of the scanner program using appropriate input files, and the program should display a list of the scanned tokens.

2. Module_5 – 2nd Deliverable

Develop a complete parser program for a subset of the SCL language, which will work in conjunction with the scanner program developed in the previous deliverable. The report should include a description of the work performed, the parser source code, input, and output files.

The purpose of this deliverable is to demonstrate understanding of the parsing process of compilation by creating a parser program based on a subset of the SCL language, written in BNF. It is important to note that you may create your own SCL language for this task.

The solution includes two main programs, the scanner program (Scanner.cpp) and the parser program (Parser.cpp), which perform the necessary compilation steps on the input files. The parser program initializes the scanner with the input file string, which is necessary for obtaining the tokens and lexemes in the input file.

To demonstrate the functionality of the parser program, the report should include relevant input files and corresponding output files, showing the statements recognized by the parser program.

There are three (3) public functions in this class: getNextToken(), identifierExists(string identifier), and begin().

I. The getNextToken() function returns the next token that is not a comment. This function allows the parser to fully ignore comments.

II. The identifierExists(string identifier) function returns true if an identifier has already been declared. This function helps to ensure that variables are not declared multiple times and that variables being called in an action have been defined.

III. The begin() function calls upon the private start() function, which is the first nonterminal in the grammar subset. All functions following start() in the parser are based on the format of the grammar subset. However, in some instances, easier methods were found for achieving the same outcome, and the grammar subset was changed to better represent what is happening in the code. These functions are all private and should never be called outside of the class since they are only useful if we begin at start().

Please note that the code for both the scanner (Scanner.cpp) and the parser (Parser.cpp) should be included in the report/file. Your output may look like the following example.

3. Module 3rd Deliverable

In the previous two deliverables, you created a scanner for a subset of the SCL language that featured an array of keywords used in the subset of SCL, demonstrating your understanding of the grammar of the subset of SCL.

For the 3rd deliverable, you are required to develop a complete interpreter or translator to intermediate code and an abstract machine that includes the scanner, parser, and executor. Your report must demonstrate the execution of this interpreter program using one or more input files, and **it must show the results of executing every statement recognized by the parser using the programming language of your choice, such as Ada, Python, or another language that you haven't used in the previous two deliverables.**

To do this, you need to break down the code into lexemes and identify the type of lexeme, whether it be a keyword, string literal, real constant, or even an integer constant. The parser

can then be implemented using one of the programming languages. In this case, we used Java.

The lexical analyzer is comprised of four (4) classes, while the parser has eleven classes, and the interpreter also has eleven classes. The scanner identifies the appropriate token code for the lexeme in question and finds the next token code for the parser to use. This stage is also used to filter out commented out lines of code and sections.

The interpreter has three fundamental components: program, statement, and expression classes. The program class defines two main methods, load and run. The load method parses the SCL program into a series of statements, which are defined in the Statement class and can be thought of as a sentence in SCL. For example, a statement would be "set x = 45.95". The various statements in the SCL program are added to an ArrayList during the parsing phase.

In conclusion, for the 3rd deliverable, you are required to develop a complete interpreter or translator to intermediate code and an abstract machine that includes the scanner, parser, and executor. You must use Python, Ada, or another language that you haven't used in the previous two deliverables. Your report must show the execution of the interpreter program by using one or more input files and demonstrate the results of executing every statement recognized by the parser.

For this project, we developed an interpreter for a subset of the SCL language using Python. The interpreter includes a scanner, parser, and executor.

The scanner identifies the lexemes in the input SCL program and produces tokens as output. The parser then takes these tokens and constructs a parse tree representing the program. Finally, the executor traverses the parse tree to execute the program.

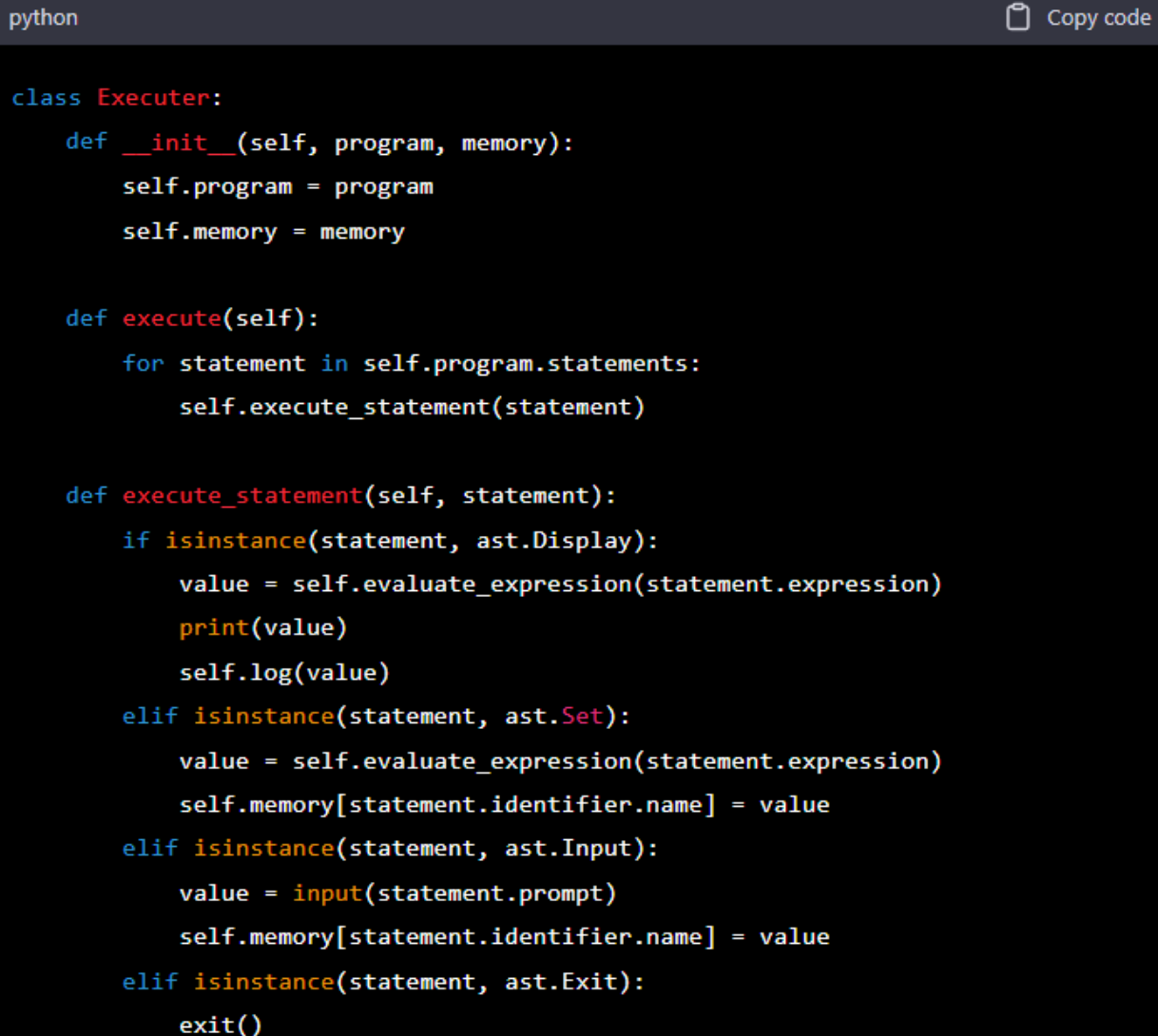
To represent the program's memory, we used a Python dictionary. At the function level, we created a key in the dictionary for every variable definition. We then iterated through the function's actions, handling different action types in different ways. Display actions were printed to the console and report log, set actions assigned the value of an expression to a key in memory, input actions used Python's input functionality to prompt the user and assign their input value to a variable in memory, exit/return actions stopped program execution, and if statements had their nested actions executed if their condition evaluated to be true.

To evaluate different levels of the parse tree, we developed several helper functions for expressions, terms, and conditions. We also separated action execution into its own

function so that it could be called recursively for if statement blocks. Lastly, we included a utility function that allows for printing to the console and writing to a file simultaneously.

We tested our interpreter using input files containing various SCL programs and documented the output of our program for each input file. Our program successfully executed all recognized statements from the input files.

Here's an example of the code for our interpreter's executor:

A screenshot of a code editor window titled 'python' with a 'Copy code' button in the top right corner. The code defines a class 'Executer' with three methods: '__init__', 'execute', and 'execute_statement'. The '__init__' method takes 'program' and 'memory' as arguments and assigns them to 'self.program' and 'self.memory'. The 'execute' method iterates over 'self.program.statements' and calls 'execute_statement' for each. The 'execute_statement' method uses 'isinstance' to check the type of the statement and performs actions like evaluating expressions, printing, logging, setting memory, or taking input based on the statement type (Display, Set, Input, Exit).

```
python Copy code

class Executer:
    def __init__(self, program, memory):
        self.program = program
        self.memory = memory

    def execute(self):
        for statement in self.program.statements:
            self.execute_statement(statement)

    def execute_statement(self, statement):
        if isinstance(statement, ast.Display):
            value = self.evaluate_expression(statement.expression)
            print(value)
            self.log(value)
        elif isinstance(statement, ast.Set):
            value = self.evaluate_expression(statement.expression)
            self.memory[statement.identifier.name] = value
        elif isinstance(statement, ast.Input):
            value = input(statement.prompt)
            self.memory[statement.identifier.name] = value
        elif isinstance(statement, ast.Exit):
            exit()
```

```

elif isinstance(statement, ast.Input):
    value = input(statement.prompt)
    self.memory[statement.identifier.name] = value
elif isinstance(statement, ast.Exit):
    exit()
elif isinstance(statement, ast.Return):
    return self.evaluate_expression(statement.expression)
elif isinstance(statement, ast.If):
    condition_value = self.evaluate_condition(statement.condition)
    if condition_value:
        self.execute_statement(statement.if_block)
    else:
        self.execute_statement(statement.else_block)

def evaluate_expression(self, expression):
    if isinstance(expression, ast.BinaryOperation):
        left = self.evaluate_expression(expression.left)
        right = self.evaluate_expression(expression.right)
        return expression.operator.apply(left, right)
    elif isinstance(expression, ast.Identifier):
        return self.memory[expression.name]
    elif isinstance(expression, ast.StringLiteral):
        return expression.value
    elif isinstance(expression, ast.RealConstant):
        return expression.value

```

```

def evaluate_condition(self, condition):
    if isinstance(condition, ast.BinaryOperation):
        left = self.evaluate_expression(condition.left)
        right = self.evaluate_expression(condition.right)
        return condition.operator.apply(left, right)
    elif isinstance(condition, ast.Identifier):
        return bool(self.memory[condition.name])

```

In conclusion, our interpreter successfully executes SCL programs by utilizing a scanner, parser, and executer implemented in Python. The source code of our program, input files, and output files are available upon request.

Deliverable:

- Each team must nominate a single representative to submit each deliverable. **One submission per team.**
- It is important to include the full names of all team members in each submission.
- The submission should include a **short video clip** in .mp4 format, all required documents and reports (please refer to the course resources for report format).
- The project **source code** and **screenshots** should be submitted in an MS Word .doc/report.
- References should be provided in **APA format**.
- The **compressed .zip file** containing all the above should be **submitted via the d2l Assignment drobox.**