Group member

YU QIANSHUO : 1210027225
WANG KUN : 1210025783
ZHENG AO : 1220005554

Group member contribution
YU QIANSHUO : symbol_table(.cpp & .h)
WANG KUN, ZHENG AO: analyzer(.cpp & .h)
ZHENG AO: driver(.cpp)

Compile
gcc version 8.0 or above is required for the code to compile.
` g++ token.cpp dfa.cpp scanner.cpp parser.cpp parsetree.cpp parser_print.cpp symbol_table.cpp analyzer.cpp driver.cpp -o analyzer -std=c++17 `

Run
`./analyzer test_correct.pyc`
`./analyzer test_wrong.pyc`

Analyzer Summary

symbol_table.cpp
symbol_table.cpp implements the functionality for managing a symbol table in a compiler. It includes the implementation of the PycSymRef and PycSymbol classes, which represent references to symbols and the symbols themselves, respectively. The SymbolTable class manages a hierarchy of symbol tables, allowing for the lookup of symbols by name. The lookup method searches for a symbol in the current table and, if not found, recursively searches in the parent table. The check_local method checks if a symbol is defined in the current table. The print_table method prints the contents of the symbol table, including symbol names, types, declaration lines, and reference lines, in a formatted manner. This functionality is crucial for semantic analysis in the compiler, ensuring that symbols are correctly identified and their scopes are managed.

symbol_table.h
symbol_table.h defines the symbol table for a compiler. It includes PycSymRef for tracking symbol references and PycSymbol for representing symbols with their declaration, type, and references. The SymbolTable manages a hierarchy of tables, enabling scoped symbol lookups with lookup and check_local methods. It also features print_table for debugging. This header is essential for semantic analysis, ensuring proper symbol management during compilation.

analyzer.cpp

s_analyser.cpp implements semantic analysis for a compiler. It includes the SemanticAnalyser class, which performs tasks such as error reporting, symbol table generation, and type assignment. The reportError method logs errors with line numbers. The genSymbolTableTop and genSymbolTable methods create and populate the symbol table, handling declarations, function definitions, and parameter lists. The assignTypes method traverses the abstract syntax tree (AST) to assign types to expressions, ensuring type consistency and reporting mismatches. This class is essential for ensuring that the program's syntax adheres to semantic rules, preparing the AST for further processing like code generation.

analyzer.h
s_analyser.h defines the SemanticAnalyser class for performing semantic analysis in a compiler. It includes methods for generating a symbol table, assigning types to expressions, and reporting errors. The genSymbolTableTop and genSymbolTable methods create and populate the symbol table, handling declarations and function definitions. The assignTypes method traverses the abstract syntax tree (AST) to ensure type consistency, reporting mismatches. The reportError methods log errors with line numbers. This class is crucial for validating the program's syntax against semantic rules and preparing the AST for code generation.

driver.cpp
sa_driver.cpp is the main driver for the semantic analysis phase of a compiler. It integrates the scanner, parser, and semantic analyzer. The run_scanner_and_parser function checks if the input file exists, runs the scanner to tokenize the file, and processes the token list to handle newlines. It then uses the Parser to parse the tokens into an abstract syntax tree (AST). If parsing succeeds, it initializes a SemanticAnalyser to generate the symbol table and assign types to expressions. It prints the symbol table and the AST if no errors occur. The main function either takes a filename from the command line or prompts the user for input. This file orchestrates the flow from source code to a semantically analyzed AST.

####running result

Two files test_correct.pyc and test_wrong.pyc are prepared. Their specific contents are as follows:

```
int a;
int main(){
    a = 0;
    if(a > 0){
        aa = 0;
    }
    else{
        aa = 1;
    }
    return 0;
    while(a < 0){
        aa = aa * aa + aa + 1;
    }
};
```

```
1   int a;
2   int main(){
3       a = 0
4
5   }
```

The result of analyzer on test_correct.pyc is :

```
PS D:\compiler\compiler-main\compiler-main\Analyzer> ./analyzer test_correct.pyc
>>
Parser: Start
Parsing Successfully


TableID   Name               Type               DeclLine  Refs
-------------------------------------------------------------
0         Semantic analysis succeeded.
```

When the analyzer is run on test_correct.pyc, the process begins with the scanner, which tokenizes the input source code by reading it line by line and converting it into a sequence of tokens. These tokens represent keywords, identifiers, operators, and other syntactic elements, allowing the source code to be broken down into manageable components. Next, the parser takes these tokens and attempts to parse them into an abstract syntax tree (AST) by following predefined grammar rules for the programming language. The parser successfully parses the input, generating an AST that reflects the program's structure. After parsing, the semantic analyzer checks the program for logical consistency, ensuring that symbols (like variables) are declared and used correctly. It generates a symbol table that tracks variable scope and types. While the semantic analysis typically flags undeclared variables, the provided output shows that the analysis was successful, with no semantic errors reported. The result includes a successful parsing message and semantic analysis success, indicating that the symbol table was generated, even though the code contains an undeclared symbol (aa), suggesting potential limitations in the error detection.

The result of analyzer on test_correct.pyc is :

```
Parser: Start
terminate called after throwing an instance of 'std::runtime_error'
  what():  Syntax Error: Expecting ';' at line 5
```

When the analyzer is run on test_wrong.pyc, the scanner tokenizes the input code, breaking it down into individual components such as keywords, identifiers, and operators. The parser then attempts to construct an abstract syntax tree (AST) from these tokens. However, during the parsing process, it encounters a syntax error due to a missing semicolon at the end of the line a = 0. The parser expects a semicolon to terminate the statement, but its absence triggers a runtime error. This results in the program throwing a std::runtime_error exception with the message "Syntax Error: Expecting ';' at line 5". This indicates that the parser has correctly identified a syntax issue in the code, specifically the failure to include the required semicolon, which is a fundamental syntax rule in many programming languages.