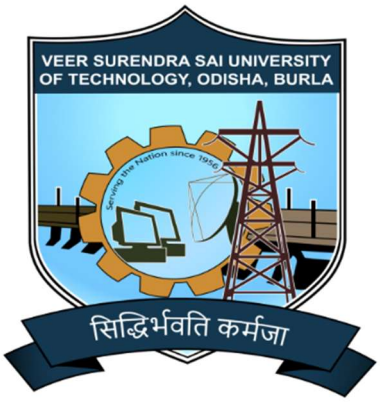
PROJECT REPORT ON
UniC COMPILER FOR C LANGUAGE

Submitted By

Satyajit Pradhan (2102081044)

Under the Expert Guidance of

Dr. D.C. Rao



DEPARTMENT OF INFORMATION TECHNOLOGY,

VEER SURENDRA SAI UNIVERSITY OF TECHNOLOGY

BURLA, SAMBALPUR

CONTENTS

INTRODUCTION

UniC is a compiler for the C programming language written in Python. The project aims to provide a fast, efficient, and reliable tool for developers working with the C language. The compiler is currently in the development stage, with only the frontend of the compiler having been built. The frontend includes the symbol table manager, lexer, parser, and semantic analyser.

The lexer is responsible for breaking the source code into tokens, which are the smallest units of meaning in the language. The parser takes these tokens and uses them to build an abstract syntax tree (AST), which represents the structure of the program. The semantic analyser then checks the AST for errors and generates intermediate code. The symbol table manager keeps track of all the symbols (variables, functions, etc.) used in the program and ensures that they are used correctly.

The UniC compiler is designed to be simple and easy to use, with a focus on providing a good developer experience. The compiler has a simple and intuitive command-line interface, and it provides clear and helpful error messages when issues are encountered.

The UniC compiler is written in Python, which was chosen for its simplicity, readability, and wide availability. Python is a popular language for compiler development, and it provides a number of libraries and tools that can be used to simplify the development process. Additionally, Python is a high-level language, which makes it easier to write and maintain the compiler code.

In summary, UniC is a compiler for the C programming language written in Python. The frontend of the compiler, including the lexer, parser, semantic analyser, and symbol table manager, has been built. The compiler is designed to be simple, easy to use, and open-source, with a focus on providing a good developer experience. The compiler is written in Python, which was chosen for its simplicity, readability, and wide availability.

MOTIVATION

The motivation for the UniC compiler project stems from the need for a simple and easy-to-use compiler for the C programming language. While there are many compilers available for C, such as GCC and Clang, they can be complex and difficult to use for beginners. Additionally, many of these compilers are closed-source, which can make it difficult for developers to customize and extend them.

The UniC compiler aims to address these issues by providing a simple, easy-to-use compiler that is also customizable. The compiler is designed to be easy to understand and modify, making it an ideal tool for teaching compiler design and development. Additionally, the compiler is written in Python, which is a popular language for compiler development and is widely used in the academic community.

The UniC C compiler project was initiated as a part of the Compiler Design course at our university. The main objective of the project is to provide us with hands-on experience in the inner workings of an actual compiler. Through this project, we aim to apply the concepts learned in class to a real-world compiler development project. Additionally, the project also provides us with the opportunity to work in a team and gain experience in developing a large software project.

In summary, the motivation for the UniC compiler project is to provide a simple, easy-to-us compiler for the C programming language that can be used for teaching compiler design and development. The project was initiated as a part of a university course on compiler design and aims to provide students with hands-on experience in compiler development and team-based software development.

OBJECTIVES

The main objective of the UniC C compiler project is to provide a comprehensive understanding of the inner workings of a compiler and gain practical experience in implementing lexical analysis, syntax analysis, and code generation. The project aims to achieve this by developing a functional compiler for the C programming language, with a focus on the frontend components.
The specific objectives of the project are as follows:

1. Implement a symbol table manager that can efficiently store and manage symbols used in the program. The symbol table manager should be able to handle different types of symbols, such as variables, functions, and labels, and provide fast lookup and insertion operations.
2. Develop a lexer that can tokenize the source code and provide a stream of tokens to the parser. The lexer should be able to handle different types of tokens, such as keywords, identifiers, and operators, and provide accurate and consistent tokenization.
3. Implement a parser that can parse the token stream generated by the lexer and build an abstract syntax tree (AST) that represents the structure of the program. The parser should be able to handle different types of statements and expressions, and generate a well-formed AST that can be used by the semantic analyser.
4. Develop a semantic analyser that can analyse the AST generated by the parser and check for semantic errors. The semantic analyser should be able to handle different types of semantic errors, such as type mismatches and undeclared variables, and provide clear and helpful error messages.

5. Write the compiler in Python, which is a popular language for compiler development and is widely used in the academic community. The compiler should be well-structured, modular, and easy to understand and maintain.
6. Work in a team and gain experience in developing a large software project. The project provides an opportunity to work in a team and gain experience in project management, communication, and collaboration.

Through this project, we aim to delve deep into the intricate stages of compilation, starting from lexical analysis, where we break down the source code into meaningful tokens, followed by syntax analysis, where we establish the structure of the code through parsing techniques. As we progress, we will tackle the challenging task of semantic analysis, ensuring that the code adheres to the language's rules and constraints.

By achieving these objectives, the UniC C compiler project aims to provide a deep understanding of the inner workings of a compiler and the different components that make up a compiler. The project also aims to provide hands-on experience in compiler development and the opportunity to apply the concepts learned in class to a real-world project.

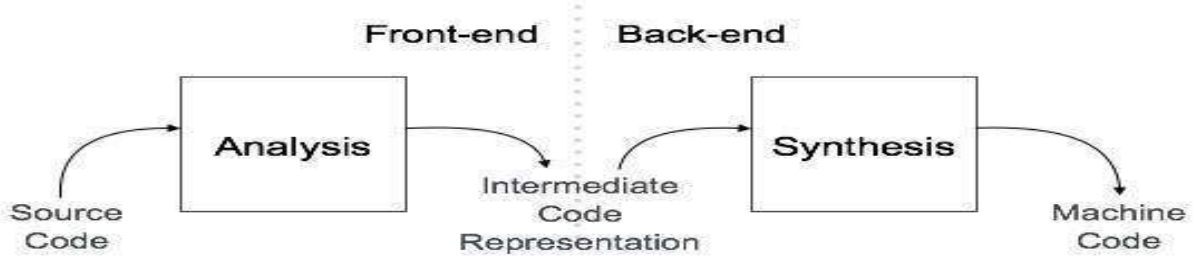
References:

- Engineering a Compiler – Cooper & Torozon
- TinyC GitHub Repository - https://github.com/Aniket025/TinyC-Compiler
- Introduction to Compiler Design - GeeksforGeeks
- https://medium.com/@pasi_pyrro/how-to-write-your-own-c-compiler-from-scratch-with-python-90ab84ffe071

IMPLEMENTATION

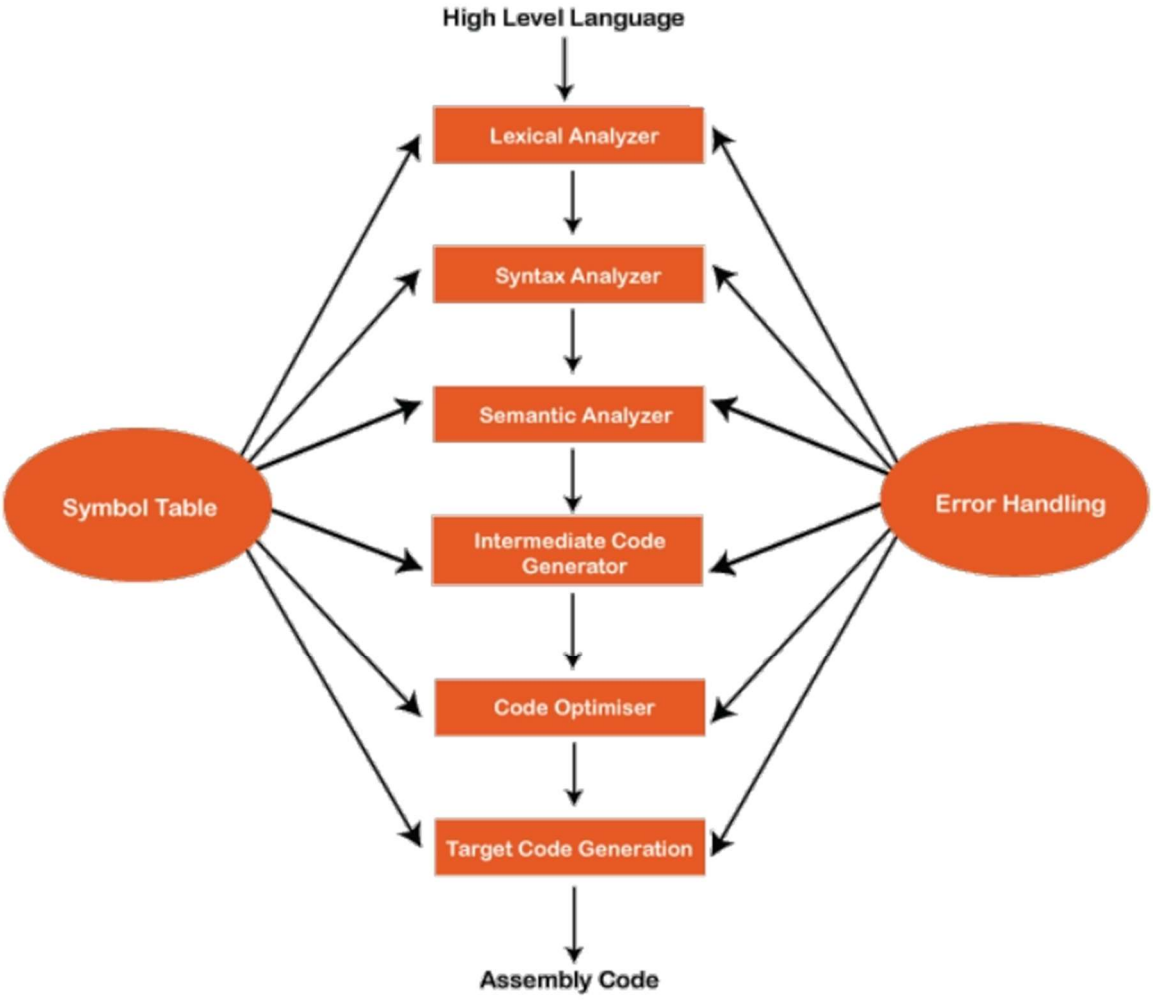
The compiler was implemented by dividing each phase of the compiler into separate modules. This modular approach allowed us to work on different phases of the compiler simultaneously, and made it easier to test and debug the compiler.

Analysis part of compiler breaks the source program into constituent pieces and imposes a grammatical structure on them which further uses this structure to create an intermediate representation of the source program. It is also termed as front end of compiler.



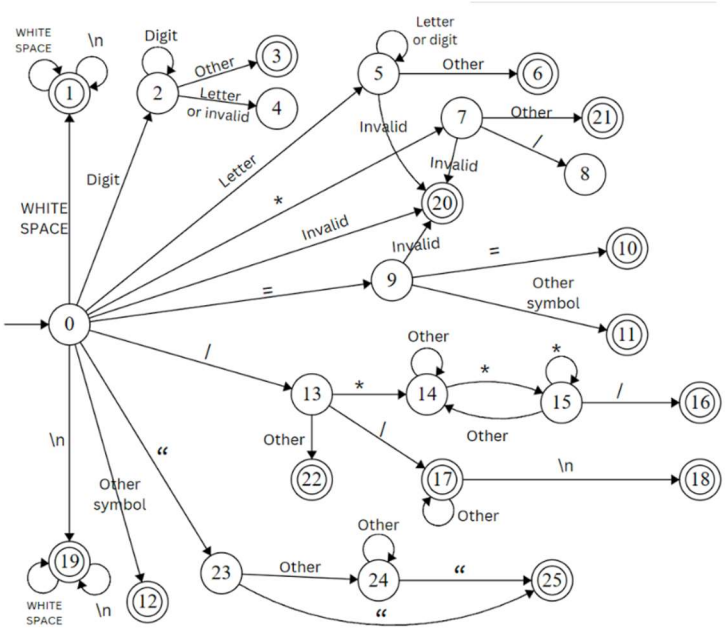
Phases of Compiler:

Each phase of a compiler is responsible for a specific task of the compilation process. The following is a brief explanation of the main phases of a compiler implemented in this project:



1. Lexical Analysis: -

The first phase of the compiler is the lexical analysis phase, which is responsible for breaking the source code into tokens. The lexer module was implemented using a deterministic finite automaton (DFA), which was designed using a state transition diagram. The DFA takes the source code as input and outputs a stream of tokens, which are then passed to the next phase of the compiler.



State	Token type
1,19	White space
3	Number
6	Identifier or keyword
10,11 12,21	Symbol
16,18	Comment
25	String Literal

DFA for Lexer of C language

Pseudocode for lexer:

1. Initialize the lexer with the input file name
2. Open the input file and read the first chunk of source code into a buffer
3. Initialize the current line number and the first line number
4. Initialize an empty list of tokens for each line number
5. Initialize an empty list of lexical errors
6. Initialize the current state of the DFA to the initial state
7. While there is still source code in the buffer:
   a. Get the next character from the buffer
   b. Use the current state and the character to determine the next state using the DFA transition table
   c. If the current state is a final state:
      i. Determine the token type based on the current state
      ii. Add the token to the list of tokens for the current line number
      iii. If the token is an identifier, add it to the symbol table

iv. Reset the current state to the initial state.

d. If the end of the buffer is reached and there is still more source code to read:

i. Read the next chunk of source code into the buffer

ii. Continue processing the buffer from the next character

iii. Repeat until the input file is completely scanned

8. If there are any lexical errors, report them to user.

2. Syntax Analysis: -

The second phase of the compiler is the syntax analysis phase, which is responsible for parsing the token stream and building an abstract syntax tree (AST). The parser module was implemented using an LL(1) parsing table, which was generated from a grammar of the C language. The parser takes the token stream as input and outputs an AST, which represents the structure of the source code.

Grammer Specifications:

- Program -> Declaration-list
- Declaration-list -> Declaration Declaration-list | EPSILON
- Declaration -> Declaration-initial Declaration-prime
- Declaration-initial -> #SA_SAVE_MAIN #SA_SAVE_TYPE Type-specifier #SA_SAVE_MAIN #SA_ASSIGN_TYPE ID
- Declaration-prime -> #SA_ASSIGN_FUN_ROLE Fun-declaration-prime | #SA_ASSIGN_VAR_ROLE #SA_MAIN_POP Var-declaration-prime
- Var-declaration-prime -> #SA_ASSIGN_LENGTH ; | [#SA_ASSIGN_LENGTH NUM] ;
- Fun-declaration-prime -> (#SA_INC_SCOPE #SA_SAVE_MAIN Params #SA_ASSIGN_FUN_ATTRS) #SA_MAIN_CHECK Compound-stmt #CG_CALC_STACKFRAME_SIZE #CG_RETURN_SEQ_CALLEE #SA_DEC_SCOPE
- Type-specifier -> int | void
- Params -> #SA_SAVE_TYPE #SA_SAVE_PARAM int #SA_ASSIGN_TYPE ID #SA_ASSIGN_PARAM_ROLE Param-prime Param-list | void Param-list-void-abtar
- Param-list-void-abtar -> ID Param-prime Param-list | EPSILON
- Param-list -> , #SA_SAVE_PARAM Param Param-list | EPSILON
- Param -> Declaration-initial #SA_ASSIGN_PARAM_ROLE Param-prime
- Param-prime -> #SA_ASSIGN_LENGTH [ ] | #SA_ASSIGN_LENGTH EPSILON
- Compound-stmt -> { Declaration-list Statement-list }

- Statement-list -> Statement Statement-list | EPSILON

- Statement -> Expression-stmt | Compound-stmt | Selection-stmt | Iteration-stmt | Return-stmt | Switch-stmt
- Expression-stmt -> Expression #CG_CLOSE_STMT ; | #SA_CHECK_WHILE #CG_CONT_JP continue ; | #SA_CHECK_BREAK #CG_BREAK_JP_SAVE break ; | ;
- Selection-stmt -> if (Expression) #CG_SAVE Statement else #CG_ELSE Statement #CG_IF_ELSE
- Iteration-stmt -> #SA_PUSH_WHILE while #CG_LABEL #CG_INIT_WHILE_STACKS (Expression) #CG_SAVE Statement #CG_WHILE #SA_POP_WHILE
- Return-stmt -> return Return-stmt-prime #CG_SET_RETVAL #CG_RETURN_SEQ_CALLEE
- Return-stmt-prime -> ; | Expression ;
- Switch-stmt -> #SA_PUSH_SWITCH switch (Expression) { Case-stmts Default-stmt } #SA_POP_SWITCH
- Case-stmts -> Case-stmt Case-stmts | EPSILON
- Case-stmt -> case NUM : Statement-list
- Default-stmt -> default : Statement-list | EPSILON
- Expression -> Simple-expression-zegond | #SA_CHECK_DECL #SA_SAVE_FUN #SA_SAVE_TYPE_CHECK #CG_PUSH_ID ID B
- B -> = Expression #SA_TYPE_CHECK #CG_ASSIGN | #SA_INDEX_ARRAY [Expression] #SA_INDEX_ARRAY_POP H | Simple-expression-prime
- H -> = Expression #SA_TYPE_CHECK #CG_ASSIGN | G D C
- Simple-expression-zegond -> Additive-expression-zegond C
- Simple-expression-prime -> Additive-expression-prime C
- C -> #CG_SAVE_OP Relop Additive-expression #SA_TYPE_CHECK #CG_RELOP | EPSILON
- Relop -> < | ==
- Additive-expression -> Term D
- Additive-expression-prime -> Term-prime D
- Additive-expression-zegond -> Term-zegond D
- D -> #CG_SAVE_OP Addop Term #SA_TYPE_CHECK #CG_ADDOP D | EPSILON
- Addop -> + | -
- Term -> Factor G
- Term-prime -> Factor-prime G
- Term-zegond -> Factor-zegond G
- G -> * Factor #SA_TYPE_CHECK #CG_MULT G | EPSILON
- Factor -> (Expression) | #SA_CHECK_DECL #SA_SAVE_FUN #SA_SAVE_TYPE_CHECK #CG_PUSH_ID ID Var-call-prime | #SA_SAVE_TYPE_CHECK #CG_PUSH_CONST NUM
- Var-call-prime -> #SA_PUSH_ARG_STACK (Args #SA_CHECK_ARGS) #CG_CALL_SEQ_CALLER #SA_POP_ARG_STACK | Var-prime
- Var-prime -> #SA_INDEX_ARRAY [Expression] #SA_INDEX_ARRAY_POP | EPSILON
- Factor-prime -> #SA_PUSH_ARG_STACK (Args #SA_CHECK_ARGS) #CG_CALL_SEQ_CALLER #SA_POP_ARG_STACK | EPSILON
- Factor-zegond -> (Expression) | #SA_SAVE_TYPE_CHECK #CG_PUSH_CONST NUM
- Args -> Arg-list | EPSILON
- Arg-list -> #SA_SAVE_ARG Expression Arg-list-prime
- Arg-list-prime -> , #SA_SAVE_ARG Expression Arg-list-prime | EPSILON

Pseudocode for Syntax Analyzer:

1. Initialize the parser with the input file name
2. Create a new instance of the lexer and semantic analyser
3. Initialize an empty list to store syntax errors
4. Create a root node for the parse tree with the name "Program"
5. Initialize the stack with two nodes: a "$" node and the root node
6. While the stack is not empty:
   a. Get the next token from the lexer
   b. Check the top of the stack and get the current node
   c. If the current node is an action symbol for the semantic analyser:
      i. Call the semantic_check method of the semantic analyser
      ii. Pop the current node from the stack
      iii. If the current node is "#SA_DEC_SCOPE" and the token is "ID":
7. Update the symbol table with the current lexeme
8. Replace the token with the updated lexeme
   d. If the current node is an action symbol for the code generator:
      i. Pop the current node from the stack
      ii. If the current node is a terminal:
      iii. If the current node matches the token:
9. If the current node is "$":
   a. Break the loop
   b. Set the token of the current node to the string representation of the token
   c. Pop the current node from the stack
   d. Get the next token from the lexer
10. Otherwise:
    a. Set the error flag of the symbol table manager to True
    b. If the current node is "$":

i. Break the loop

ii. Otherwise:

11. Append a syntax error to the list of syntax errors
12. Pop the current node from the stack
13. Set the clean_up_needed flag to True

f. Otherwise (the current node is a non-terminal):

i. Look up the parsing table to determine which production to use

ii. Get the right-hand side (rhs) of the production

iii. If "SYNCH" is in the rhs:

14. Set the error flag of the symbol table manager to True
15. If the token is "$":

a. Append a syntax error for an unexpected end-of-file

b. Break the loop

c. Otherwise:

i. Append a syntax error for a missing construct

ii. Remove the current node from the parse tree

iii. Pop the current node from the stack

iv. Otherwise (if "EMPTY" is in the rhs):

16. Set the error flag of the symbol table manager to True
17. Append a syntax error for an illegal token
18. Get the next token from the lexer

iv. Otherwise:

19. Pop the current node from the stack
20. For each symbol in the rhs:

a. If the symbol is not an action symbol:

i. Create a new node with the symbol as the name

ii. Set the parent of the new node to the current node

iii. Append the new node to the list of new nodes

b. Otherwise:

i. Create a new node with the symbol as the name

ii. Append the new node to the list of new nodes

21. For each node in the list of new nodes (in reverse order):

a. If the node is not an "EPSILON" node: Append the node to the stack

22. Call the eof_check method of the semantic analyser

23. If the clean_up_needed flag is True:

a. Clean up the parse tree by removing non-terminals and unmet terminals from leaf nodes

24. Show the list of syntax errors to the user

3.Semantic Analysis: -

The third phase of the compiler is the semantic analysis phase, which is responsible for checking the semantic correctness of the source code. The semantic analyser module was implemented using a set of semantic routines, which were designed to check for common semantic errors, such as type mismatches and undeclared variables. The semantic analyser takes the AST as input and outputs a list of semantic errors, if any.

Pseudocode for Semantic Analyzer:

1. Define the SemanticAnalyser class with the following methods:
    - __init__: Initialize the semantic checks dictionary, semantic stacks dictionary, main_found flag, main_not_last flag, arity_counter, while_counter, switch_counter, fun_param_list, fun_arg_list, semantic_errors list, and semantic_error_file.
    - scope: Return the current scope index.
    - semantic_errors: Return a string containing all semantic errors.
    - _get_lexim: Return the lexeme of the given token.

- save_semantic_errors: Write all semantic errors to the semantic_error_file.
- inc_scope_routine: Increment the current scope by appending the current size of the symbol table to the scope_stack.
- dec_scope_routine: Decrement the current scope by popping the last index from the scope_stack and updating the symbol table to only include symbols up to that index.
- save_main_routine: Save the lexeme of the current token to the main_check stack.
- pop_main_routine: Pop the last two elements from the main_check stack.
- save_type_routine: Save the type of the current token to the type_assign stack and set the declaration_flag to True.
- assign_type_routine: If the current token is an ID and the type_assign stack is not empty, assign the type of the current symbol to the top element of the type_assign stack, update the type_assign stack to include the current symbol index, and set the declaration_flag to False.
- assign_fun_role_routine: If the type_assign stack is not empty, assign the role of the current symbol to "function" and update the address of the current symbol in the memory manager.
- assign_param_role_routine: Assign the role of the current symbol to "param".
- assign_var_role_routine: Assign the role of the current symbol to "local_var" or "global_var" depending on the current scope.
- assign_length_routine: If the current token is a NUM and the type_assign stack is not empty, assign the arity of the current symbol to the value of the current token. If the current token is a "[", assign the type of the current symbol to "array".

- save_param_routine: Save the type of the current parameter to the fun_param_list.
- push_arg_stack_routine: Push an empty list to the arg_list_stack.
- pop_arg_stack_routine: Pop the last element from the arg_list_stack if it is not empty.
- save_arg_routine: Save the type of the current argument to the last element of the arg_list_stack.
- assign_fun_attrs_routine: Assign the arity and params attributes of the current function symbol based on the fun_param_list and reset the fun_param_list.
- check_main_routine: Check if the current function symbol matches the signature of the main function and update the main_found and main_not_last flags accordingly.
- check_declaration_routine: If the current token is an ID and the type of the current symbol is not defined, raise a semantic error.
- save_fun_routine: Save the index of the current function symbol to the fun_check stack.
- check_args_routine: Check if the arguments of the current function call match the parameters of the current function symbol and raise a semantic error if they do not.
- push_while_routine: Increment the while_counter.
- check_while_routine: If the current token is "continue" and the while_counter is zero, raise a semantic error.
- pop_while_routine: Decrement the while_counter.
- push_switch_routine: Increment the switch_counter.
- check_break_routine: If the current token is "break" and the while_counter and switch_counter are both zero, raise a semantic error.
- pop_switch_routine: Decrement the switch_counter.

- save_type_check_routine: Save the type of the current operand to the type_check stack.
- index_array_routine: If the current token is "[" and the type_check stack is not empty, set the top element of the type_check stack to "int".
- index_array_pop_routine: If the type_check stack is not empty, pop the top element from the type_check stack.
- type_check_routine: If the type_check stack is not empty, pop the top two elements from the type_check stack and check if their types match. If they do not, raise a semantic error.
- semantic_check: Call the appropriate semantic routine based on the given action symbol and input token.
- eof_check: Check if the main function was found and raise a semantic error if it was not.

2. Pass the input through every single method of the SemanticAnalyser class.
3. If there is an error, report to the user.
4. Otherwise, move to intermediate code generation phase.

Symbol Table Manager: -

A symbol table is a data structure used in compilers to store information about the symbols (variables, functions, etc.) in a program. It is used during the semantic analysis phase of compilation to check for syntax errors and to assign memory locations to symbols. The symbol table typically includes information such as the name, type, scope, and storage class of each symbol. It may also include additional information such as the value of constants and the parameters of functions. The symbol table is used by the code generator to generate machine code that correctly accesses the memory locations assigned to each symbol.

Pseudocode for Symbol Table Manager:

1. Define a class SymbolTableManager with the following methods:
    - __init__: Initialize the class variables _global_funcs, scope_stack, temp_stack, arg_list_stack, symbol_table, declaration_flag, and error_flag.
    - init: Initialize the class variables scope_stack, temp_stack, arg_list_stack, symbol_table, declaration_flag, and error_flag.
    - scope: Return the current scope index.
    - insert: Append a new row to the symbol_table with the given lexim and current scope.
    - _exists: Check if a row with the given lexim and scope already exists in the symbol_table.
    - findrow: Search the symbol_table for a row with the given value and return it. If no row is found, return None.
    - findrow_idx: Search the symbol_table for a row with the given value and return its index. If no row is found, return None.
    - install_id: If the declaration_flag is False, search the symbol_table for a row with the given lexim and return its index. If the declaration_flag is True, append a new row to the symbol_table with the given lexim and current scope, and return its index.
    - get_enclosing_fun: Search the symbol_table for the enclosing function at the given level and return it. If no function is found, return None.
2. Define the class variable _global_funcs as a list of dictionaries representing the built-in functions of the compiler.

3. Define the class
   methods scope, insert, _exists, findrow, findrow_idx, install_id,
   and get_enclosing_fun.
4. Define the instance methods __init__ and init.
5. Initialize the SymbolTableManager class by calling the init method.

CONCLUSION

In conclusion, the UniC C compiler project was a success. We were able to design and implement a functional compiler for the C programming language, with a focus on the frontend components. The compiler is able to accurately parse and analyse source code written in the C language, and generate intermediate code that can be used for further processing.

The symbol table manager, lexer, parser, and semantic analyser components of the compiler all function as intended, and the compiler is able to handle a wide range of C language constructs.

One of the main challenges we faced during the development of the project was ensuring that the compiler can handle the complex semantics of the C language. However, through careful design and implementation of the semantic analyser, we were able to overcome this challenge.

Another challenge was ensuring that the compiler is well-structured, modular, and easy to understand and maintain. Through regular code reviews and by following best practices in software development, we were able to ensure that the compiler is of high quality and can be easily extended and modified in the future.

In terms of future work, there are several areas where the compiler could be improved. For example, the current implementation of the compiler only generates Abstract Syntax Tree, and does not include an intermediate code generator or code optimizer. Adding these components would allow the compiler to generate executable code, and improve the performance of the generated code.

Additionally, the current implementation of the compiler only supports a limited subset of the C language. Expanding the compiler to support additional language features, such as pointers and dynamic memory allocation, would make it more useful for a wider range of applications.

The development of this compiler has provided us with valuable experience in software development, problem-solving, and project management. We have gained a deep understanding of the inner workings of a compiler and the complex semantics of the C language. We have also learned how to work effectively as a team, and how to communicate and collaborate effectively to achieve a common goal.

We are proud of the final product that we were able to create, and we believe that it provides a solid foundation for future work in the field of compiler development. We are confident that the skills and knowledge we have gained through this project will serve us well in our future studies and careers in the field of computer science.