

## **Abstract**

Compilers are extremely important tools because they form the core of the infrastructure on which other software is built. The purpose of this project is to learn the techniques and tools in compiling, to understand the principles of interpreters and compilers from a language designer's point of view, and gain a deeper understanding of the nature of programming languages. The specific approach of this project is to use Python to implement a subset of Matrix Lab (MATLAB), a programming language and software implementation officially defined and published by MathWorks. The achievements of this project including the implementation of most of the operators, several statements, different data types, and some matrix operations and built-in functions, which are considered an important advantage of MATLAB. As an important feature of the interpreter/compiler, our interpreter also implements error handling, reporting the different types of errors detected when there are errors in the input code. In addition, we have an elaborately designed test module capable of testing all the features mentioned above, using manually designed test cases and auto-generated scripts to demonstrate the correctness of the results produced by our product or review bugs to repair.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Literature Review</b>	<b>4</b>
2.1	The principle of interpreters and compilers . . . . .	4
2.2	The development of compilers . . . . .	5
2.3	The development of interpreters . . . . .	6
2.4	Interpreters compilers for popular modern programming language . . . . .	7
2.4.1	Python interpreters . . . . .	7
2.4.2	JAVA interpreters . . . . .	8
2.4.3	C/C++ compilers . . . . .	8
<b>3</b>	<b>Project Description</b>	<b>8</b>
<b>4</b>	<b>Work Division</b>	<b>9</b>
4.1	Project Structure . . . . .	9
4.2	Han Sanyue's work . . . . .	9
4.3	Zou Yang's work . . . . .	9
4.4	Hao Ting ting's work . . . . .	9
<b>5</b>	<b>Han Sanyue's Individual Part</b>	<b>10</b>
5.1	Declaration of Authorship . . . . .	10
5.2	Description of all the work finished . . . . .	10
5.3	Lexical Analysis . . . . .	10
5.3.1	Introduction . . . . .	10
5.3.2	Tool . . . . .	10
5.3.3	Token Class . . . . .	11
5.3.4	Algorithm . . . . .	11
5.3.5	Example . . . . .	12
5.4	Syntax Analysis . . . . .	13
5.4.1	Introduction . . . . .	13
5.4.2	Node Class . . . . .	13
5.4.3	Tool . . . . .	13
5.4.4	Algorithms . . . . .	14
5.4.5	Example . . . . .	15
5.5	Semantic Analysis . . . . .	15
5.5.1	Introduction . . . . .	15
5.5.2	Interpreter . . . . .	15
5.5.3	Data Types . . . . .	16
5.6	Results . . . . .	18
<b>6</b>	<b>Zou Yang's Individual Part</b>	<b>19</b>
6.1	Declaration of Authorship . . . . .	19
6.2	Preparation . . . . .	19
6.3	Compiler back-end section . . . . .	19
6.3.1	Back-end code logic (code can be seen in the directory "IV_assgen") . . . . .	19
6.3.2	Back-end code conclusion . . . . .	20
6.4	Error handling in the front-end . . . . .	20
6.4.1	Construction of the exception class (code can be seen in the folder "exceptions")	20
6.4.1.1	Construction of the parent class "InterpretException" and the inheritance process . . . . .	21

6.4.1.2	Construction of the parent class "InterpretException2" and the inheritance process . . . . .	21
6.4.2	Construction of sample error files (files can be found in the folder "error_handling") . . . . .	22
6.4.3	Invocation of exception classes in three phases (The process is contained in the code of the three phases) . . . . .	23
6.4.3.1	Lexical analysis phase ("lexer.py") . . . . .	23
6.4.3.2	Syntactic phase (parser.py) . . . . .	24
6.4.3.3	Semantic analysis phase (operations.py, interpreter.py, utils.py) . . . . .	26
6.4.4	Test work for error handling (test_error_handling.py) . . . . .	27
6.4.5	Error Handling Conclusion . . . . .	27
<b>7</b>	<b>Hao Tingting's Individual Part . . . . .</b>	<b>30</b>
7.1	Declaration of Authorship . . . . .	30
7.2	Preparation . . . . .	30
7.2.1	The importance of testing in our project . . . . .	30
7.2.2	Why we use MATLAB for comparison . . . . .	30
7.2.3	Something about testing . . . . .	31
7.3	Description of my work . . . . .	31
7.3.1	Manual testing . . . . .	31
7.3.2	Problems identified and solved . . . . .	31
7.3.2.1	Solve the problem of symbols and operators in arrays . . . . .	31
7.3.2.2	Diverse floating-point representation . . . . .	32
7.3.2.3	Solve matrix addition and subtraction operation and representation problems . . . . .	33
7.3.3	Automatic test . . . . .	33
7.3.3.1	Automatically compare test results . . . . .	33
7.3.3.2	Randomly generate test code:stage 1 and stage 2 . . . . .	34
7.4	Results . . . . .	36
7.5	Discussion and future work . . . . .	36
7.6	Conclusion . . . . .	37
<b>8</b>	<b>Overall Discussion and Conclusions . . . . .</b>	<b>38</b>
8.1	The limitation on functionalities . . . . .	38
8.2	The limitation on error reporting . . . . .	38
8.3	The limitation on performance . . . . .	38
<b>9</b>	<b>Suggestions for possible future work. . . . .</b>	<b>38</b>

# 1 Introduction

To design an interpreter or a compiler, we can provide another version of software implementation for an existing programming language or create a new programming language whose syntax is defined by ourselves. We chose to develop a MATLAB implementation and imitate its features and features because of the existing programming language. MATLAB is an interpreted language that is easy to realize than those compiled languages such as C/C++. We can find its syntax rules files written in normal form, which can be very useful for beginners like us who have no experience developing interpreters or compilers. From the perspective of normal users, text output is similar to the official software, especially Matlab\_R2021a. There are two general approaches to developing this project. The first is to read the explanation in the official documentation to understand the design behavior of the different syntax. The second is to run some experimental code on MATLAB software and then see their results. In other words, like reverse engineering, some detailed features are simulated by extrapolating from the internal design of the official software and then finding some equivalent in Python if the documentation is not detailed enough. Moreover, for an existing programming language, it is convenient for us to carry out tests. We could execute the test scripts using the existing interpreter or compiler and compare our results.

In this thesis, we will first introduce how compilers and interpreters work, review the development history of interpreters and compilers and introduce the interpreter or compilers of some famous modern programming languages like C/C++, Java, and Python in Section 2. Then we will briefly introduce the structure of our project in section 3 and the work division among group members in section 4. After that, in section 5,6,7, they are personal part, we can describe our tasks respectively in detail. Section 8 contains an overall discussion in which we will compare our product with the official MATLAB software and discuss the advantages and disadvantages and finally conclude in section 9.

## 2 Literature Review

### 2.1 The principle of interpreters and compilers

Programming languages are notations for describing a flow of computational actions to people and machines. However, according to their corresponding instructions sets, the CPUs only know how to execute machine code consisting of 0s and 1s, not those higher-level programming languages that we are familiar with, such as C, Java, or Python. Therefore, before a high-level program can be run, it must be converted into a form that the computer can execute[16].

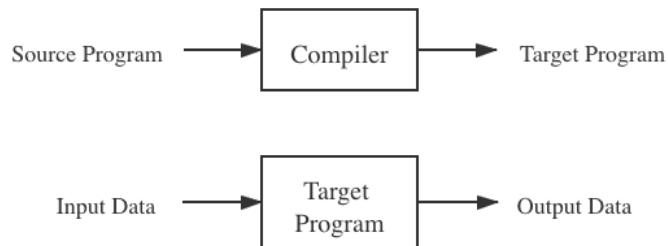


Figure 1: The principle of compilers

There are two different ways to realize this. The first one is to use a compiler to read a program in one language and then translate it into an equivalent program in another language[25]. If the target program is an executable machine-code program, the user can then be called to process inputs and produce outputs. Figure 1 shows how compilers work.

The other kind of language processor is called an interpreter. Instead of producing an executable target program as a translation, an interpreter directly executes the operations specified by the source

program. Figure 2 shows the principle of an interpreter.

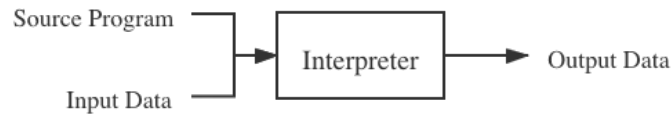


Figure 2: The principles of interpreters

The advantage of a target program produced by a compiler over an interpreter at mapping inputs into outputs is its speed. In contrast, an interpreter can give better error diagnostics than a compiler because it executes the program statement by statement[27]. As figure[?] shows, compilers behave differently in many perspectives.

	Compiled language	Interpreted language
Running speed	Fast (binary)	Slow (execute while explaining)
Portability (cross-platform)	Bad(CPU instruction system changes and errors occur)	Good(Take the interpreter with you)
Update	Recompile	Explain only the updated content
Safety	Good (no need to provide source code)	Poor (delivered together with source code)

Figure 3: Differences between the compiler and interpreter

## 2.2 The development of compilers

Complex and ubiquitous software architectures underpin the global economy[18]. Compilers and high-level languages are the cornerstones of such software. In addition, powerful and elegant compilation technology also has an extremely important value in hardware synthesis and other fields. Compilers and high-level languages are central to the information age as semiconductor technology [15].

By far, the most striking achievement in the compiler field is the widespread use of high-level languages[?]. From banking and enterprise management software to high-performance computing and various World Wide Web (Web) applications, the vast majority of today’s software is written in high-level languages and compiled statically or dynamically [7].

The first practical computer compiler, written for the A-0 system in 1952 by American female computer scientist Grace Murray Hopper on UNIVAC I, compiled programs into machine code. However, its function was much closer to what we now know as a linker or loader. The compiler loads punched cards that carry programs written in human languages into the computer. The computer spits out another set of cards containing machine code. The second set of cards is loaded into the computer, and the computer can execute the new program. In response to the shortcomings of assembly language, Fortran, led in 1957 by John Warner Backus, an American computer scientist working for IBM, was the first fully functional high-level language compiler implemented[11].

In the 1950s, the field of the compiler was beginning, and scientists’ research focus was limited to the conversion from high-level languages to machine code and the time and space requirements of optimizing programs [8]. Since then, much new knowledge has emerged in program analysis and transformation, code automation, and runtime services. At the same time, compilation algorithms are also used to facilitate software and hardware development, improve application performance, detect or avoid software defects and malware. At the same time, the field of compilation is increasingly intersecting with other areas such as computer architecture.

Object-oriented and data abstraction languages were first introduced in the late 1960s and early 1970s, and it was realized that these concepts could greatly improve programmer productivity. Computer architects and compiler writers first began to consider the static and dynamic optimizations invented in the field of parallelism compilers, seeing parallelism as a promising solution to the problem of computers not being fast enough. Instruct level parallelism was introduced in Seymour Cray's CDC 6600 and CDC 7600 and IBM System/360 Model 91. The compiler front end greatly benefited from developing systematic theories based on the automata theory of lexical, syntactic analysis techniques[20].

In the 1970s, CRAY-1 became the world's first commercially successful supercomputer [39], and the success of the CRAY machine was largely due to the introduction of the vector register. Like scalar registers, vector registers allow programs to perform many small data vectors [24]. Cray Research has also developed a very aggressive vectorization compiler. It is similar in many ways to the earlier TI compilers, but the Cray compiler has some features that make it very interesting. One of the most important features is that the ability to provide compiler feedback to the programmer. Many projects have focused on the automatic generation of other parts of the compiler, including code generation. In short, the Cray programmers finally achieved the three goals of performance, productivity, and portability.

In the early stage of developing a high-level language compiler, the technology is not mature enough, and the generated target code is large, the execution efficiency is low. These problems affect the promotion and application of high-level language. Cocke(1976) made an in-depth study of compiler code generation technology and put forward a series of optimization methods[2]. For example, the program integration, loop conversion, the elimination of common subexpressions, code movement, register positioning, storage unit reuse, so that the quality of the compiler has been greatly improved. The development of compilation technology reached a new stage[3].

In the 1990s, the messaging library was replaced by the Messaging Interface (MPI), and all those vendor-specific parallelization instructions were replaced by OpenMP. More scalable parallel systems emerged, such as Thinking Machines CM-5 and others with thousands of commercial microprocessors. SIMD instruction sets for single-chip microprocessors are on the market, and there is growing interest in redirectable compilers with SIMD support [18], such as SSE and SSE2 from Intel.

Shortly after 2000, multi-core microprocessors began to be widely offered by many vendors. Compilers include applications for more complex algorithms that are used to infer or simplify information in programs. Compilers are increasingly a part of window-based interactive development environments. They include editors, linkers, debuggers, and project managers.

Compared to earlier compiler implementations, today's compilation algorithms are significantly more complex. While early compilers used simple and intuitive techniques for lexing programs, today's lexing techniques are based on formal languages and automata theory, making the compiler front end more systematic [22]. Similarly, the reconfigurable compiler work that used simple and intuitive techniques for dependency analysis and loop transformation today uses powerful algorithms based on integer linear programming.

Despite all the advances in compiler technology, some people still see compilers as more of a problem than a solution. They want the predictability that comes from the compiler, rather than the advanced analysis and code transcoding that optimizes the compiler in the background.

## 2.3 The development of interpreters

A compiled language means that after we write a program, we translate the code into a binary file and execute the program by executing the binary file. Interpreted languages, on the other hand, do not convert binaries but compile them when needed. The interpreter includes a compilation process, but this compilation process does not generate object code. A Python interpreter consists of a compiler that converts source code into bytecode, which is then executed line by line through the Python virtual machine and a virtual organization. When we write Python code, we get a text file with an .py extension that contains Python code. To run the code, a Python interpreter is required to implement Python files. In 1989, Guido began writing a compiler for the Python language; In

1991, the first Python compiler was created. It is implemented in C and can call C library files. Python already has a module-based extension system with classes, functions, exception handling, and core data types. Since then, Python has been updated as shown in Figure 4, with the current version being Python 3.8. In summary, Python 2.x is legacy; Python 3.x is the present and future of the language.

Version	Date	Annotation
Python 1.0	January 199	Added lambda, map, filter and reduce
Python 2.0	October 16, 2000	Added a memory recovery mechanism, which forms the basis of the current Python language framework
Python 2.4	November 30, 2004	
Python 2.5	September 19, 2006	
Python 2.6	October 1, 2008	
Python 2.7	July 3, 2010	
Python 3.0	December 3, 2008	
Python 3.1	June 27, 2009	
Python 3.2	February 20, 2011	
Python 3.3	September 29, 2012	
Python 3.4	March 16, 2014	
Python 3.5	September 13, 2015	
Python 3.6	December 23, 2016	

Figure 4: Python historical versions

## 2.4 Interpreters compilers for popular modern programming language

Here is a principle of how modern programming language processors are realized. Some of them belong to the interpreter, like Python, some of them belong to the compiler, like C/C++, while there are also some kinds of language whose execution go through both a stage of compilation and a stage of interpreting, like Java.

### 2.4.1 Python interpreters

There are some popular python interpreters: CPython, Jython, IronPython, and Pypy.

1. CPython is a reference implementation written in C and is the most widely used Python interpreter.
2. IPython is an interactive interpreter based on CPython. In other words, IPython is only enhanced in terms of interaction, but executing Python code is the same as that of CPython[26].
3. Jython and IronPython are compilers that generate bytecode for the underlying virtual machine, and are primarily two alternative implementations for the Java virtual machine[9].
4. Pypy is another Python interpreter, and Pypy is the most outstanding effort to develop an alternative interpreter to improve Python's execution speed. Its goal is the speed of execution.

PyPy can significantly improve the execution speed of Python code because Python code is compiled dynamically using JIT technology[1].

### 2.4.2 JAVA interpreters

For Java, the compiler compiles the Java source files (.java files) into bytecode files (.class files), which are binary. This bytecode is the "machine language" of the JVM (a virtual machine capable of running Java bytecode). Javac.exe can be thought of simply as a Java compiler. The Java interpreter is part of the JVM. The Java interpreter is used to interpret programs that have been compiled by executing the Java compiler. Java.exe can be thought of simply as a Java interpreter. The JVM interprets the execution of a bytecode file as the JVM manipulates the Java interpreter to interpret the execution of a bytecode file. The JVM masks information specific to a specific operating system, allowing Java programs to run unchanged on multiple platforms by generating only the object code (bytecode) that runs on the Java Virtual Machine.

### 2.4.3 C/C++ compilers

Visual C++ is the most popular compiler on the Windows platform which integrates well with VS and has good compilation efficiency and compiled code efficiency. GCC/G+ is the preferred option on Linux/ UNIX platforms and supports N component platforms.

## 3 Project Description

Let us look into the working process of interpreters or compilers in detail. We could find that they operate as a sequence of stages, each transforming one representation of the original program into another.

A typical decomposition is shown in Figure 5.

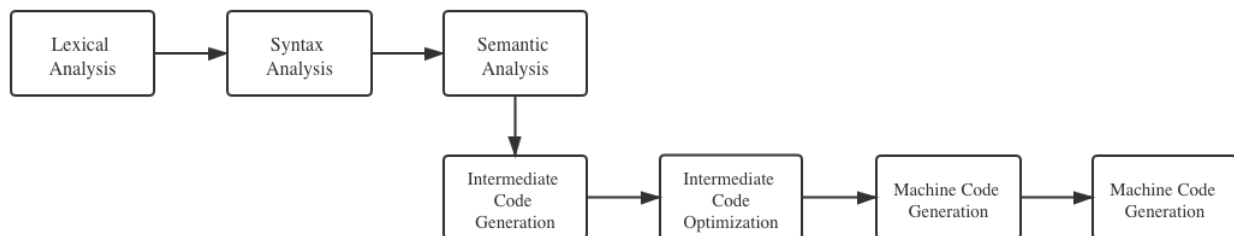


Figure 5: Phases of a compiler

In this flow chart, there are seven steps in total. The three steps above, namely lexical analysis, syntax analysis, and semantic analysis, are usually regarded as the front end, which is responsible for analyzing the source program by breaking it up into constituent pieces, create an abstract syntax tree to represent it, and report the problem properly when bugs are detected. To develop an interpreter, we only need to implement these three parts.

The four steps below, namely intermediate code generation, intermediate code generation, machine code generation, and machine code optimization, are usually regarded as the back end, which is responsible for synthesis the target program using the information provided by the front end. The back end is also necessary for developing a compiler.



## 4 Work Division

### 4.1 Project Structure

The structure of our final project is shown in Figure 6, in which the responsibility of different members are labeled accordingly.

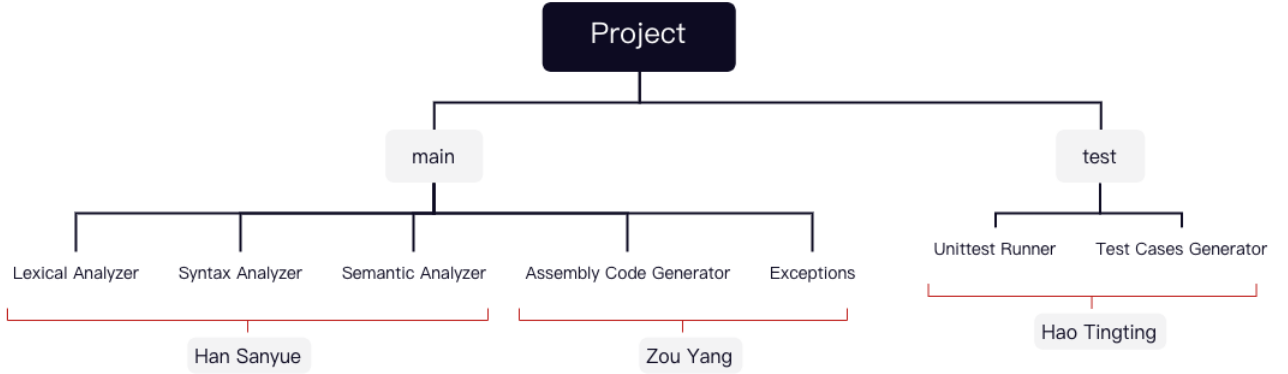


Figure 6: Structure of our project

### 4.2 Han Sanyue's work

Han Sanyue developed the lexical analyzer, syntax analyzer, semantic analyzer in the front end. He integrated these part so that the interpreter could work in two different modes. The feature realized including arithmetic operators, logical operators, relational operators for evaluation of expressions, selection statements and iteration statements for control the flow of execution, four basic data types for representation of different kinds of data, and a few built-in functions that provide scientific calculation functionality just like MATLAB.

### 4.3 Zou Yang's work

The four stages in the back end is assigned to Zou Yang at first. In our early plan the back end combined with the achievements from front end could form a compiler. Zou Yang realized some very basic function for the back end, namely some arithmetic operation. However, with the project's progress, it seems that this part is too complicated to continue, the reason will be explained in Zou Yang's part. So, after a discussion our group reassigned some work involves the error reporting in front end to Zou Yang, He finished this part perfectly, so the interpreter could report different kinds of error in the three stages, print not only the type of error but also the position of the code that cause the error.

### 4.4 Hao Ting ting's work

Hao Tingting undertakes the work of testing, including both testing on individual modules and product. In this part, by choosing Matlab as the comparison tester, automatic testing is realized. In addition, this part also implements a test generator which can randomly generate several test combinations. And some output rules of MATLAB were found and used to solve some bugs, such as symbols and operators in array, array and floating point results representing diversity to ensure that the interpreter or compiler will not only produce the right results for correct code but also report the proper error for buggy codes.

## 5 Han Sanyue's Individual Part

### 5.1 Declaration of Authorship

I declare that all of the following are true:

- 1) I fully understand the definition of plagiarism.
- 2) I have not plagiarized any part of this project and it is my original work.
- 3) All material in this report is my/our own work except where there is clear acknowledgement and appropriate reference to the work of others.

### 5.2 Description of all the work finished

As mentioned earlier, my main work involves realizing the three phrases in front end, namely lexical analysis, syntax analysis, and semantic analysis, in other words the modules an interpreter needed. To interpret a program, the lexical analyzer firstly convert the input code from a string to a list of tokens. Then the syntax analyzer will build an abstract syntax tree to represent the structure of the program. Finally, the semantic analyzer will visit the syntax tree recursively to performing the corresponding operations. Integrating these modules we will get an interpreter. In the following subsections, I will explain the input, the output, the functionality, algorithms and data structures applied, the code design of these modules in details firstly, and then summarize the achievements.

### 5.3 Lexical Analysis

#### 5.3.1 Introduction

The lexical analyzer is responsible for the conversion from a sequence of characters to a list of tokens. The source code written in the script is essentially some text that is made up of a sequence of characters. The character could be encoded using ASCII or some other encoding method such as Unicode. In order to simplify complexity, let's firstly assume that the input code is always encoded using ASCII, so there are no more than 128 kinds of different characters that may exist in our input text stream. Just like reading a book, although the content is printed character by character, people read them by recognizing the characters as groups called word. For a programming language processor, the first thing it does on the input character sequence is to split the characters into similar groups that are called tokens. The tokens have different types. Some tokens belong to the keyword type, such as "if", "else", "while", "for" which are very common to see in lots of languages, some tokens belong to identifier type, that is the case when you declare to define a variable whose name is up to you, some tokens belong to literal types, it occurs when you write some constant numbers or strings in the code, and some tokens belong to operator types, such as the plus symbol "+", assignment symbol "=", etc. Let's see an example of analyzing a code fragment, the expression "age >= 35", in figure 6.

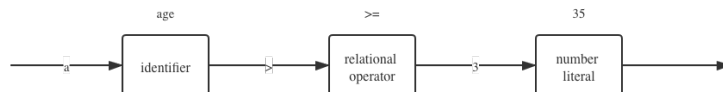


Figure 7: Lexical Analysis Diagram

#### 5.3.2 Tool

To recognize the symbol, there are two different methods. One is to implement the finite state machine, the algorithm is to transform between states depend on every input character, and the final stopping state indicates the token types. Another method is to use regular repression, which

is adopted in this part. There are two advantages of regular expression over finite state machines on this task. The first one is that a finite state machine is difficult to implement because the data structure of the finite state machine and the transformation condition between different stages need to be coded in detail very carefully. However, if regular expression is applied, for Python, in which we develop our project, it has a very powerful built-in module called `re` that is capable of compiling a pattern you designed and automatically tested if a pattern matches with a string. The second one is that regular expression is the basis of the Extended Backus-Naur Form, which will be explained in the syntax analysis part.

In the following some examples of the patterns used for recognize tokens are introduced.

The pattern for recognizing the keyword type looks like this:

```
1 "break | case | catch | continue | elseif | else | end | for | function"
```

It means that if the words split by a "—" is match with the character stream, a keyword is recognized.

The regular expression for identifier looks like this:

```
1 "[a-zA-Z]+[a-zA-Z0-9_]*"
```

This means that a valid identifier in MATLAB should start with a letter and could be followed by a number of letter, digit, or underline.

You may wonder there is a problem: all the keywords could be matched by the pattern of identifier, so why won't the lexical analyzer be confused about whether a token should belong to identifier type or keyword type? The answer is easy: when recognizing the next token in the input string, we use all the patterns defined in a specific order. If any one of them matches the string, then the token is recognized. Therefore, to make the lexical analyzer work correctly, it is necessary to not only write the correct regular expressions but also make sure the order of the patterns is correct. For every situation when one kind of token is the prefix of another token, the shorter token should appear later than the longer token in order to avoid mistakes. For example, the operator "`;`" should appear later than "`;`", and "`=`" should appear later than "`==`", etc.

### 5.3.3 Token Class

A class is defined for the tokens. There are four attributes in this class:

- type

Type indicating what kind of token it is that is distinguished by different regular expressions. This information is important in the next stage, where the token list is passed to a parser, and the parser analyzes the structure of the code based on the types of a number of continuous tokens.

- text

Text is the string matched in the input stream. This information will always be needed in the following stages

- col

The column number where this token appears in the source code, which is useful in error reporting

- row

The row number where this token appears in the source code, which is useful in error reporting

### 5.3.4 Algorithm

The core of the lexical analyzer is a function that reads from the input string, recognizes tokens, and removes used characters until nothing is left in the input sequence. If at any time the beginning of the input sequence could not be matched by any pattern, the analyzer will raise an error. The flow chart representing this algorithm is shown in Figure 8.

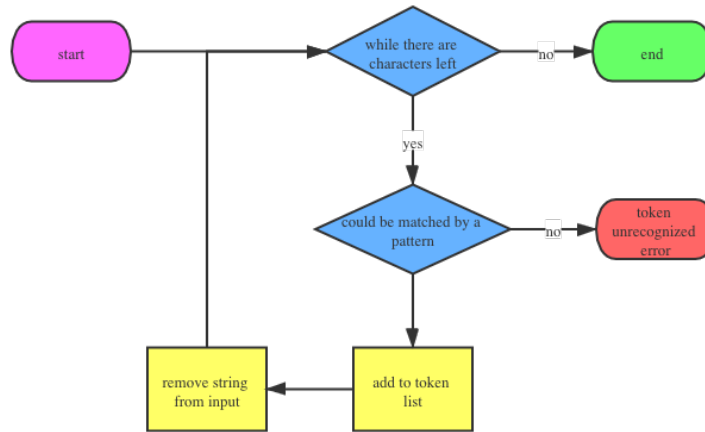


Figure 8: Lexical Analysis Diagram

### 5.3.5 Example

Let's see an example of the lexical analysis. Here is a very simple script.

```

1 a = 0;
2 while a < 10
3     a
4     a = a + 1;
5 end

```

We can print its lexical analysis result, which is a list of tokens.

1	row = 1	col = 0	type = IDENTIFIER	text = 'a'
2	row = 1	col = 2	type = ASS	text = '='
3	row = 1	col = 4	type = NUMBER_LIT	text = '0'
4	row = 1	col = 5	type = EO_STMT	text = ';'
5	row = 1	col = 6	type = EO_STMT	text = '\n'
6	row = 2	col = 0	type = KEYWORD	text = 'while'
7	row = 2	col = 6	type = IDENTIFIER	text = 'a'
8	row = 2	col = 8	type = REL	text = '<'
9	row = 2	col = 10	type = NUMBER_LIT	text = '10'
10	row = 2	col = 12	type = EO_STMT	text = '\n'
11	row = 3	col = 4	type = IDENTIFIER	text = 'a'
12	row = 3	col = 5	type = EO_STMT	text = '\n'
13	row = 4	col = 4	type = IDENTIFIER	text = 'a'
14	row = 4	col = 6	type = ASS	text = '='
15	row = 4	col = 8	type = IDENTIFIER	text = 'a'
16	row = 4	col = 10	type = ADD	text = '+'
17	row = 4	col = 12	type = NUMBER_LIT	text = '1'
18	row = 4	col = 13	type = EO_STMT	text = ';'
19	row = 4	col = 14	type = EO_STMT	text = '\n'
20	row = 5	col = 0	type = KEYWORD	text = 'end'
21	row = 5	col = 3	type = EO_STMT	text = '\n'

It is shown that for every token, the text, the type, the col number, and the row number and all recorded.

## 5.4 Syntax Analysis

### 5.4.1 Introduction

The syntax analyzer is responsible for the conversion from a token list to an abstract syntax tree. The reason why the conversion is needed is that a list is a linear structure. It is not complex enough to represent the logic in expressions or statements. For example, let's see an example in figure 9. This is an expression containing operators with different precedence. Only if the structure of the abstract syntax tree is known will the interpreter know the order of evaluation. In this example, the tree will be visited in a post-order traversal, the interpreter will visit the node containing "3" and "5" firstly, and then using the operator "\*" to evaluate a multiplication operation, and finally use the result of the "\*" node and the value in the node containing "2" as operand, use the operator "+" to perform an addition operation.

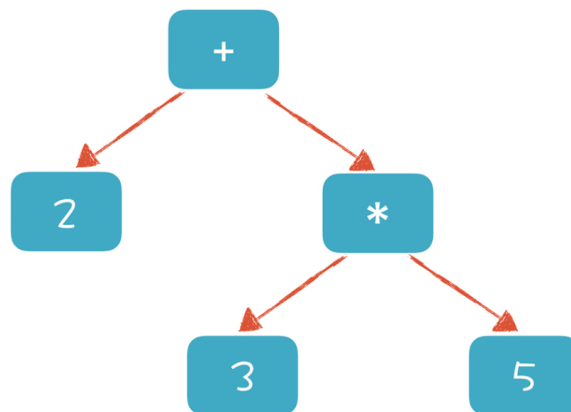


Figure 9: Abstract Syntax Tree

Now in our project, there are 18 different types of the abstract syntax tree node. Some of them represent a type of statement node. For example, EXPSTMT stands for an expression statement, SELSTMT stands for an if statement or a switch statement, ITRSTMT stands for a while statement or a for statement. Some of them represent a type of expressions, such as number literal or operator expression.

### 5.4.2 Node Class

Similar to the previous stage, in syntax analyzer, we also need to define a class to represent the node of an abstract syntax tree. There are three attributes in the node:

- type

Similar to the token class, in node class there is also a attribute to record the type of the node, which will be used in the execution of the code.

- text

The text is the same string that the tokens carried, but now they belongs to the node. In the previous abstract syntax tree example, the number literal "2", "3", "5", and operator "+", "\*" are all text in nodes.

- children

This is a list that containing the children nodes.

### 5.4.3 Tool

In the previous session, we talked about regular expression. A regular expression is very powerful, and it can represent a wide range of string patterns. However, one deficiency restricted its usage in the

syntax analyzer: the regular does not support recursive expression. In the grammatical definition of code, there are at least two kinds of recursion. The first one is at the expression level: one expression could always be a part of a bigger expression, just like the example we saw. The second one is at the statement level: one statement could always be a part of a bigger statement because for some flow controlling statement, like the "if" statement, one component for these kinds of statement is a code block that contains multiple other statements.

Since regular expressions cannot be used to represent the logical structure of codes, a more powerful tool called Extended Backus–Naur Form is implemented to deal with the definition of grammar. Extended Backus–Naur Form is a context-free grammar, which means that it is able to represent recursive patterns. Here are some of the grammar rules our project realized written in EBNF:

```

1  ass_exp ::= id '=' cln_exp
2  cln_exp ::= lor_exp ( ':' lor_exp ) *
3  lor_exp ::= lan_exp ( ( '|' | '||' | '|||' ) lan_exp ) *
4  lan_exp ::= eql_exp ( ( '&&' | '&' ) eql_exp ) *
5  eql_exp ::= rel_exp ( ( '~=' | '==' ) rel_exp ) *
6  rel_exp ::= add_exp ( ( '<=' | '<' | '>=' | '>' ) add_exp ) *
7  add_exp ::= mul_exp ( ( '+' | '-' ) mul_exp ) *
8  mul_exp ::= uny_exp ( ( '*' | '/' ) uny_exp ) *
9  uny_exp ::= ( '+' | '-' | '~' ) * pri_exp
10 pri_exp ::= identifier | number_lit | string_lit | '(' cln_exp ')' | '[' array_list

```

This grammar rule tells us that a colon expression consists of one or more logic or expression, a logic or expression consists of one or more logic and expression, logic and expression consists of one or more similar expression, a similar expression consists of one or more relational expression, a relational expression consists of one or more additive expression, an additive expression consists one or more multiplicative expression, a multiplicative expression consists one or more unary expression, a unary expression consists of one or more primary expression. However, a primary expression has different cases. It could be an identifier, a number literal, or a string literal, or a colon expression encapsulated by a pair of parentheses, or a list of arrays encapsulated by brackets. Since the definition of primary expression used the predefined colon expression, this grammar rule is recursive.

And another important function of the EBNF grammar rules file is that it defines the precedence of operators. As shown in the previous example, the multiplicative operator is evaluated earlier than additive expression because, in the syntax analysis stage, the multiplicative node is parsed as a child node of the additive node. And the root cause is that in the definition of the grammar rule file, the multiplicative expression is closer to the primary expression, so parsing multiplicative expression will run before the method for parsing additive expression runs.

#### 5.4.4 Algorithms

There are two ways to build an abstract syntax tree. The first one is to use some advanced tools such as antlr, flex, and bison. These tools could use the EBNF grammar rules file to generate a syntax analyzer automatically. The second one is to write from scratch. That is the way I adopted.

The class I designed to build the abstract syntax tree is called Parser. There is so many methods implemented in the parser that it is not realistic to explain them all. The common thing is that most of them call others recursively. And most methods for parsing the operators have a very similar structure. The visualization of the method for parsing primary expression is shown in figure 10.

One feature to notice is that compare to the previous stage. It is obvious that in syntax analysis, the kinds of error could be reported, which have been marked as red, increased noticeably. The reason is simple, in lexical analysis, the data is in a linear structure. Every time the recognition has only two different results: the characters will either be recognized as some type of token or unrecognized.

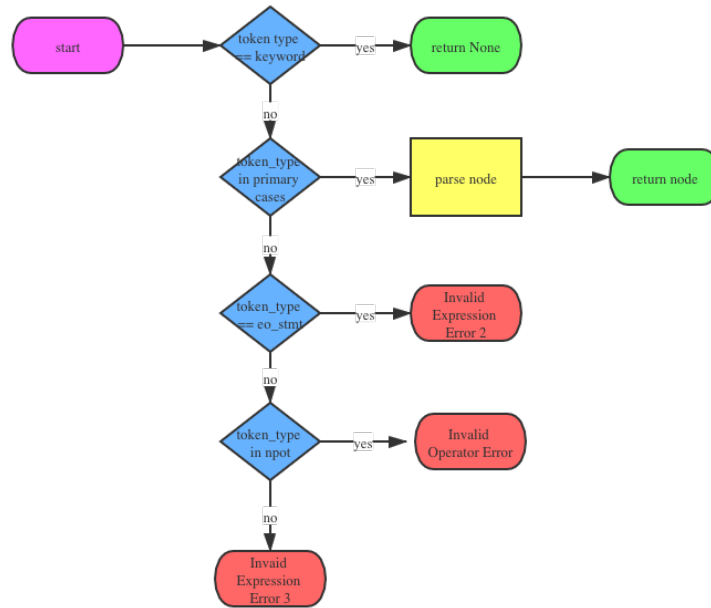


Figure 10: Primary Expression Parser

However, in syntax analysis, the dimension of the abstract syntax tree is higher than one; thus, the cases of causing error become more complicated.

#### 5.4.5 Example

Let's use the same script appeared in the lexical analysis stage as an example to see the result of after performing a syntax analysis:

```

1 a = 0;
2 while a < 10
3     a
4     a = a + 1;
5 end

```

The result is shown in figure 11.

The abstract syntax tree shows that the whole program is regarded as a statement list. Inside the statement list, there are two statements. The first one is an assignment statement that assigns number 0 to variable "a." The second one is a while statement which consists of a relational expression as loop condition and a statement list as a loop body. Inside the loop body, there is an expression statement and an assignment statement.

## 5.5 Semantic Analysis

### 5.5.1 Introduction

In the semantic analysis, the Interpreter traverses the abstract syntax tree, performing different operations on different nodes according to their node types to execute the source code. One important functionality for the semantic analyzer is to deal with the data types.

### 5.5.2 Interpreter

The Interpreter is implemented by using a class called Interpreter. It has an attribute called built-ins that stores the realization of some built-in functions. It has another attribute called variables

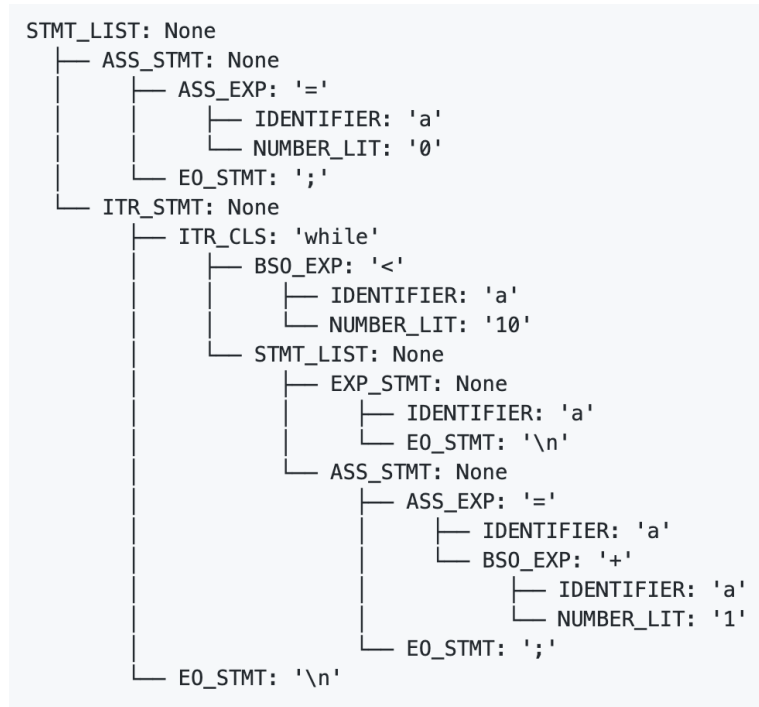


Figure 11: Abstract Syntax Tree

that represents the global variable scope. For every kind of node, there is their corresponding method for visiting them.

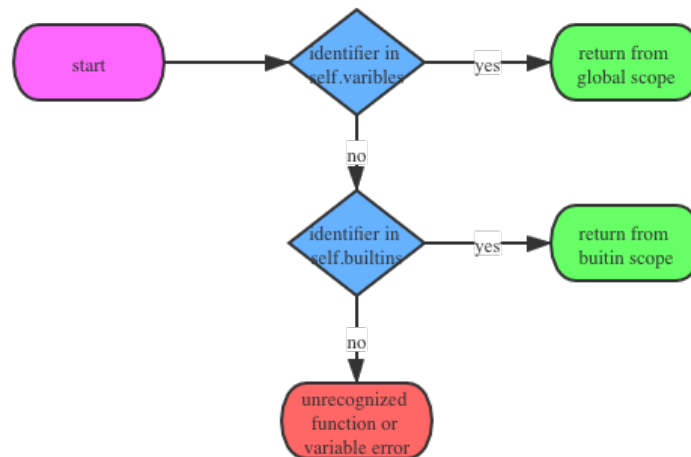


Figure 12: Interpreter Retrieving Method

In figure 12 the process of retrieving a variable is visualized. It is shown that the interpreter will firstly check its global variables scope, if a identifier is matched then return the result, otherwise check the built-ins variable scope then, if a identifier is matched then return the result, otherwise an unrecognized function or variable error will be raised.

### 5.5.3 Data Types

There are four data types implemented in our interpreter, namely, Double, Logical, String, and Char.



1. Double type is the default data type for numbers in MATLAB. It is realized based on the double precision floating point type in Python.

2. String type is created using double quotes in MATLAB. It is also realized based on the string type in Python.

3. Char type is created using single quotes in MATLAB. It is realized base on the integer type in Python.

4. Logical type is produced by the evaluation of relational expressions or logical expressions. It is realized base on the boolean type in Python.

To have a complete type system, there are three parts that a language processor should pay attention to. The first one is type inference. It means that in how could the interpreter know about the resulting the type of an expression. For some operation the type is a synthesized attribute, in which the type are inferred from descendant nodes. For example for arithmetic operation, a double type plus a double type result in a double type, while a string type plus a string type result in a string type. For other operations the type is an inherited attribute, in which the attributes are decided by ancestor nodes or itself, such as variable declaration, or logical and relational operations as mentioned earlier. The second part is type check, since it usually appears in compiled languages, in which the type of a variable will stay unchanged as long as it is initialized, but the language we implemented is interpreted, in which the type of a variable could be changed during the execution. Therefore, this part is not implemented in out project. The third part is type conversion, especially implicit type conversion. Let's see an example firstly.

```
1 >> a = ["1234" 1234]
2
3 a =
4
5     1x2 string array
6
7     "1234"     "1234"
8
9 >> b = [1234 1==1]
10
11 b =
12
13     1234         1
14
15 >> c = ["asdf" 123.4 1==1]
16
17 c =
18
19     1x3 string array
20
21     "asdf"     "123.4"     "true"
22
23 >> d = ["asdf" [123.4 1==1]]
24
25 d =
26
27     1x3 string array
28
29     "asdf"     "123.4"     "1"
30
31 >>
```

This is a demonstration of implicit type conversion using the interpreter we developed, which has the same behavior with MATLAB on this code fragments. When using brackets to create an array, the interpreter will automatically convert all of the data to a same type. In the first example a string type and a double type is put into the same array, and the double type is converted to string directly. In the second example a double type and a logical type is put into the same array, and the logical type is converted to double, mapping true to 1 and false to 0. In the third example a string type, a double type, and a logical type are put into the same array simultaneously, and the logical type is converted to string type directly, mapping true to "true" and false to "false". In the last example, a double type and a logical type is put into an array firstly and then put into an array with a string. In this case, the logical type is firstly converted to double 1, and the converted to string "1".

In our project, this feature is realized by implementing a type conversion tree in the data type system. The string type should be the root of this tree since all kinds of data could be converted into strings without losing information. Moreover, it's easy to infer that the logical type should be the child of double type. The process of converting several data into the same type is essentially an algorithm about finding the nearest common ancestor in the type conversion tree.

## 5.6 Results

The functionalities realized in this part include most of the operators in MATLAB, six kinds of statements, four data types as well as a few built-in functions.

- Operators
  - Arithmetic Operators
    - Addition
    - Subtraction
    - Multiplication
    - Division
    - Powers
    - Transpose
    - Array Sign
  - Relational Operators
  - Logical Operators
- Statements
  - expression statement
  - assignment statement
  - if statement
  - switch statement
  - for statement
  - while statement
- Data Types
  - Double
  - String
  - Char
  - Logical
- Built-in Functions

## 6 Zou Yang's Individual Part

This part of the work is done by ZOU YANG, whose UCD student number is 17205885.

### 6.1 Declaration of Authorship

I declare that all of the following are true:

- 1) I fully understand the definition of plagiarism.
- 2) I have not plagiarized any part of this project and it is my original work.
- 3) All material in this report is my/our own work except where there is clear acknowledgement and appropriate reference to the work of others.

### 6.2 Preparation

At the beginning of the project, my division of work was mainly on the back end of the compiler, using the abstract syntax tree generated on the front end to generate the corresponding assembly code, and then planning to use "GCC" to link the assembly code to generate an executable file to verify that the assembly code could be executed correctly. In the early stages of the project, the main learning resource I've been referring to is the "PlayWithCompiler" project on github. <https://github.com/RichardGong/PlayWithCompiler>.

### 6.3 Compiler back-end section

Although my division of work is for the back-end part, I have been learning the front-end part for some time, as I need to understand the composition and structure of the abstract syntax tree in order to make better use of it when generating assembly code. As stated in our proposal form, the extent of completion of the back-end is dependent on the progress of the completion of the front-end, in other words, on the progress of the abstract syntax tree produced by the front-end.

#### 6.3.1 Back-end code logic (code can be seen in the directory "IV\_assgen")

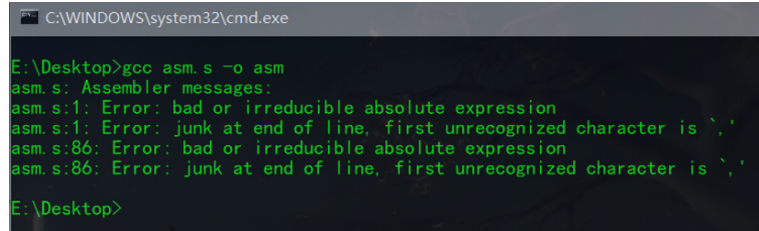
In the middle and later periods of our project, the abstract syntax tree took shape, and the basic functionality, i.e. supporting for basic variable assignment and simple operations, was initially completed in the process of lexical analysis to syntactic analysis. Therefore, I started writing code for generation of assembly code based on the abstract syntax tree produced by the front-end, which is included in the file called "asmgen.py".

For "asmgen.py", my design idea is to firstly import the abstract syntax tree generated by the front-end part, which in our project is actually a node called "ASTNode", and traversing the children of this node achieves the purpose of traversing the whole tree. Then the specific type of the node in the syntax tree should be imported, e.g. if the content of a node is "+", then the specific type is "BOP\_EXP", i.e. binary operation expression. I created a class called "Asmgen". Encapsulating the code in a class makes it easier to call it later when integrating the four parts, namely lexical analysis, syntactic analysis, semantic analysis and code generation, which can be seen in "script\_execute.py".

In the class "Asmgen", in order to make it extensible, i.e. to make it easy to modify the code in this class to generate assembly code for new operations as the abstract syntax tree supports new operations, I have adopted a top-down logic, i.e. breaking down the big problem into smaller problems. The logic for generating assembly code is as follows: the assembly code is inherently formatted with a start and end segment, which can be added directly to the final string, and then the entire syntax tree is traversed, a process that is implemented by another function which plays a leading role. A new function was used to traverse the syntax tree, in other words to traverse each node in the tree. The traversal of an individual node was done by setting up a new function, which exemplifies the idea of programming to decompose a large problem into sub-problems step by step.

### 6.3.2 Back-end code conclusion

Unfortunately, after initially writing this part of the code, which only supports variable assignment and addition and subtraction operations, the executable file I generated by using the GCC link command "gcc asm.s -o asm" only worked on Mac, but failed on Windows, which is my development environment. The error was shown in Figure 13.



```
C:\WINDOWS\system32\cmd.exe
E:\Desktop>gcc asm.s -o asm
asm.s: Assembler messages:
asm.s:1: Error: bad or irreducible absolute expression
asm.s:1: Error: junk at end of line, first unrecognized character is ','
asm.s:86: Error: bad or irreducible absolute expression
asm.s:86: Error: junk at end of line, first unrecognized character is ','
E:\Desktop>
```

Figure 13: link file using gcc raised error in windows

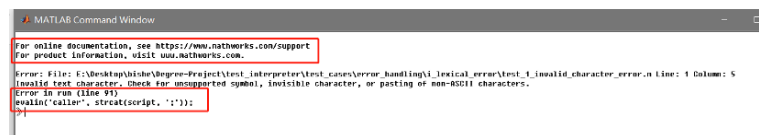
In the process of looking for a solution, I found the back-end platform compatibility problem difficult to solve. For the Windows platform, to link assembly code, I need to call the libraries underlying various operating systems. But with less than a month to submit, there was not enough time to solve this problem, so after discussing with my group members and mentor, my focus shifted from the back-end to the front-end. The final output of our project also changed from a compiler to an interpreter. For the interpreter, the platform compatibility issue is easy to solve, as there is no need to deal with the operating system underlay.

## 6.4 Error handling in the front-end

My part changed to deal with the exception throwing parts of lexical analysis, syntax analysis and semantic analysis. Specifically, we cannot assume that the user is an idealized user who cannot make mistakes. The user could inevitably write some bugs when using the interpreter to write code, which requires the interpreter to alert the user to which line, which column and what type of error has occurred.

### 6.4.1 Construction of the exception class (code can be seen in the folder "exceptions")

Python is a language that allows object-oriented programming, and the inheritance feature is very useful for the implementation of exception classes. I created two parent classes containing a certain amount of common features of the exception type. As an example, in lexical analysis there are two error types "InvalidCharacterError" and "StringTerminationError", and handling statements of MATLAB for these two exceptions are shown in Figure 14 and Figure 15. Because the redundant information above and below (the red boxed parts in Figure 1) was not essentially part of the error handling section, I ignored them here. It can be seen that the common features of both error types are that they both contain "Error: File:", followed by the path to the file, then the line and column numbers (I have marked the common features as written in Figure 14).



```
MATLAB Command Window
For online documentation, see https://www.mathworks.com/support
For product information, visit www.mathworks.com.
Error: File: E:\Desktop\bishu\Project\test_interpreter\test_cases\error_handling\lexical_error\test_1_invalid_character_error.m line: 1 Column: 5
Invalid test character, check for unsupported symbol, invisible character, or pasting of non-ASCII characters.
Error in run (line 9)
    evalin('caller','strcat(sprintf, '%')');
^
```

Figure 14: MATLAB error reporting for "InvalidCharacterError" type



Figure 15: MATLAB error reporting for "StringTerminationError" type

**6.4.1.1 Construction of the parent class "InterpretException" and the inheritance process** Based on the features stated above, I have summarized the first parent class, "InterpretException", which inherits from the generic python exception class "Exception". This allows us to redefine the "\_\_str\_\_" function to directly output the error statement we want when an exception is thrown, by setting the return value of the "\_\_str\_\_" function according to the format of the error statement ("InterpretException" is shown in Figure ??).

```
class InterpretException(Exception):
    message = {}

    def __init__(self, line=0, column=0):
        self.line = line
        self.column = column

    def __str__(self):
        return f"Line: {self.line} Column: {self.column}\n" + self.message[sys.platform]
```

Figure 16: Implementation of the "InterpretException" class

There was a hiccup here, in the process of development and continuous testing, I found that the format of the real MATLAB error statements on windows platform and Mac platform are not exactly the same. For example, if you run the same file "test\_1\_invalid\_character\_error.m" in Mac, the error message is shown in Figure 17, where you can see that the Mac platform will use a line break for a portion of an excessively long error message. Since the Python language has many easy-to-use modules, my solution was to import the module "sys", which provides a set of variables and functions for the Python runtime environment, and "sys.platform" can get the platform of the current runtime environment. [1] Simply use the Python dictionary type in the subclass to automatically select a different format for the error statement depending on the platform (the key values are "darwin" for Mac and "win32" for Windows).

<https://docs.python.org/3/library/sys.html>

```
Error: File:
/Users/hansanyue/Desktop/Degree-Project/test_interpreter/test_cases/error_handling/lexical_error/test_1_invalid_character_error.m
Line: 1 Column: 5
Invalid text character. Check for unsupported symbol, invisible character, or
pasting of non-ASCII characters.
[Error in run (line 91)
evalin('caller', strcat(script, ' '));
```

Figure 17: Format of error reporting statements for the Mac platform

After creating the parent class, the construction of the child class could be made easily by the definition of the class variable "message" in the parent class. Since the common features for lexical and syntactic analysis are contained in the "InterpretException" class, the only thing that needs to be changed for these two phases is the class variable "message"(The implementation of these exception classes can be seen in "lexical\_exception.py" and "syntactic\_exceptions.py"). For example, for the "EndMissingError" exception class in the syntactic analysis, it is very simple to implement, just inherit from "InterpretException" and redefine the class variable "message", as shown in Figure 18.

**6.4.1.2 Construction of the parent class "InterpretException2" and the inheritance process** For the exception in the semantic analysis phase, its error statements are very different from the first two

```

class EndMissingError(InterpretException):
    message = {
        'darwin': "At least one END is missing. The statement beginning here does not have a\matching end.",
        'win32': "At least one END is missing. The statement beginning here does not have a matching end."
    }

```

Figure 18: Implementation of the "EndMissingError" exception class

phases, so I created another parent class "InterpretException2". As an example, for the "UnaryOperatorForStrError" exception class in the semantic analysis, the result of MATLAB's error reporting is shown in Figure 6. Comparing this with Figure 1 or Figure 2, you can see that the error statement, which means "message", is no longer static, it needs to be adjusted according to the operator where the MATLAB code is in error. Because the type of error I was testing for is an error where the operator cannot use a minus sign, the error reporting statement in Figure 19 appears with a minus sign '-'. The second point is that it no longer contains the file path. The third point is that it only gives the line number where the error statement is located, not the column number, which makes sense as this type of error means that the whole statement is illogical. Finally, MATLAB outputs the entire error statement. By summarising the above four points, the "InterpretException2" exception class is implemented as shown in Figure 20.

The screenshot shows the MATLAB Command Window with the following text:
   
For online documentation, see <https://www.mathworks.com/support>
  
For product information, visit [www.mathworks.com](http://www.mathworks.com).
   
Unary operator '-' is not supported for operand of type 'string'.
   
Error in test\_1\_unary\_operator\_minus (line 1)
   
a="-str";
   
Error in run (line 91)
   
evalin('caller', strcat(script, ';'));
   
>

Figure 19: MATLAB's output for the "UnaryOperatorForStrError" exception

```

class InterpretException2(Exception):
    message = {}

    def __init__(self, line=0, filename='', error_statement=''):
        self.line = line
        self.filename = filename
        self.error_statement = error_statement

    def __str__(self):
        return self.message[
            sys.platform + f"Error in {self.filename} (line {self.line})\n" + self.error_statement + "\n"

```

Figure 20: Implementation of the "InterpretException2" class

For the subclasses, since "message" needs to be adapted to different situations, I have set up a function to receive parameters to change the minus '-' sign in "message" as described above, one of the implementations is shown in Figure 21. The "unary\_operator" is the variable that I have stated needs to receive the argument.

Of course, in the semantic analysis phase, there are also situations that the class variable "message" does not need to be changed, such as the exception subclass "ErrorUsingMultiply" (as shown in Figure 22). In fact it is almost identical to the exception subclasses of the first two stages, except that it inherits from a different parent class.

#### 6.4.2 Construction of sample error files (files can be found in the folder "error\_handling")

The process of constructing sample bugs is essentially to list as many errors or bugs as possible that occur when users write MATLAB code.

During the lexical analysis phase, the user may incorrectly enter special characters such as " or forget to add an end flag when defining a string or character vector, then the program should throw

```

class UnaryOperatorForStrError(InterpretException2):

    def modify_mess(self, unary_operator=''):
        UnaryOperatorForStrError.message = {
            'win32': f"Unary operator '{unary_operator}' is not supported for operand of type 'string'.\n"
        }

```

Figure 21: Implementation of the "UnaryOperatorForStrError" exception subclass

```

class ErrorUsingMultiply(InterpretException2):

    message = {
        'win32': "Error using *.\nincorrect dimensions for matrix multiplication. Check that the number of columns in the first matrix matches the number of rows in the second matrix.\n"
    }

```

Figure 22: Implementation of the "ErrorUsingMultiply" exception subclass

an exception at this stage. Therefore, I constructed the statements such as "a =" to check that the program throws the correct exception.

The exceptions that may arise during the syntactic analysis phase I summarize as the following three:

- 1-The end character "end" is missing. In MATLAB, we need to add "end" at the end of the statement blocks, such as if, while, for, otherwise we would get an error.
- 2-Incomplete statements, e.g. forgetting to add right brackets.
- 3-Invalid expressions, such as two separate statements written on one line without a separator, i.e. a semicolon, or square brackets on the left and round brackets on the right, which is a typical invalid expression.

The semantic analysis phase focuses on performing some of the advanced mathematical operations that are characteristic of the MATLAB programming language, such as multiplication and division of matrices and concatenation operations, or comparison operations on strings. For matrix operations, if the dimensions of two matrices do not conform to the rules, then the operation cannot be performed. For string comparisons, strings can only be compared with each other and not with other types such as integers, otherwise an exception is thrown.

In conclusion, these examples actually need to be refined, as there is a limit to the power of one person to exhaust all possibilities, so there may be exception types that have not been included that still need to be added.

### 6.4.3 Invocation of exception classes in three phases (The process is contained in the code of the three phases)

#### 6.4.3.1 Lexical analysis phase ("lexer.py")

The work at this stage is mainly to cut the string of the MATLAB file into individual "tokens", and how the exception class is called is shown in Figure 23.

For "CharacterVectorTerminationError", when slicing each line, if the current token is a single quote, which means that the character vector is currently in use, then determine if there is a token or space before it in the current line, and if so, then there is clearly an exception here, i.e. the character vector is not properly terminated.

After determining the type of the current token, it is common sense that if there is no error, the loop will break and continue to determine the next token on the current line. However, if after iterating through all the token types, no matching type is found, then the loop will not be jumped out by break, which means that the for loop ends normally, this situation is obviously an exception, so the for else statement block in python is used, which means that the statement in else can only be executed when the for loop ends normally. I thought of two scenarios for the exceptions that occur here. One is the case of a double quote at the beginning of a sentence, which means that there is only one single double quote on the current line, because the pair of double quotes could be recognized by the regular expression in the token type, i.e. "STRING\_LIT". Therefore, the single double quote

```

while line:
    for TYPE in TokenType:
        match = TYPE.value.match(line)
        if match:
            if TYPE == TokenType.TRA:
                if TokenType.WHITESPACE.value.sub(' ', token_text) == '':
                    raise CharacterVectorTerminationError(row, col)
            # preprocess: replace comma for space in some array construction cases
            if brackets_depth > 0 and TYPE == TokenType.ADD and token_type == TokenType.WHITESPACE:
                result_token_list.append(Token(t_type=TokenType.EO_STMT, t_text=', ', row=row, col=col - 1))
            if brackets_depth > 0 and TYPE == TokenType.WHITESPACE and token_type == TokenType.ADD \
                and result_token_list[-2].get_type() == TokenType.EO_STMT: # space - space以要删除减号前的逗号
                result_token_list.pop(-2)
            if TYPE == TokenType.L_BRACKET:
                brackets_depth += 1
            if TYPE == TokenType.R_BRACKET:
                brackets_depth -= 1

            token_text = match.group()
            token_type = TYPE
            if TYPE != TokenType.WHITESPACE and TYPE != TokenType.ANNOTATION:
                result_token_list.append(Token(t_type=TYPE, t_text=token_text, row=row, col=col))
            line = line[len(token_text):]
            col += len(token_text)
            break
        else:
            if line[0] == '\n':
                raise StringTerminationError(row, col)
            else: # line[0] in "$%"
                raise InvalidCharacterError(row, col)

```

Figure 23: Invocation of exception classes in lexical analysis

case clearly raises an exception, i.e. the string does not end properly. Another case is when the user enters an invalid character, such as "\$", which would raise an exception for an invalid text character.

#### 6.4.3.2 Syntactic phase (parser.py)

Syntactic analysis identifies the syntactic structure of a program on the basis of lexical analysis, which is a tree-like structure, also known as an abstract syntax tree. Once at this stage, exceptions about characters have been filtered out and the types of exceptions that can be detected at this stage are syntax type errors.

The first is the invalid expression exception, which is subdivided into three types because of the variety of expressions.

1) The first is the absence of a separator, such as two separate statements without a separator, or a block of statements such as if without a separator, will throw an exception, as shown in Figure 24, which determines whether the current statement is correctly separated from subsequent statements by determining whether the type of the current node is "EO\_STMT", i.e. a separator.

```

def complete_statement(self, node):
    if self.get_token_type() != TokenType.EO_STMT:
        token = self.get_token()
        raise InvalidExpressionError1(token.row, token.col)
    node.add_child(ASTNode(n_type=ASTNodeType.EO_STMT, n_text=self.tokens.pop(0).get_text()))
    return node

```

Figure 24: Invocation of the "InvalidExpressionError1" exception class

2) The second is when parsing the primary expression and another operand is found to be missing, then obviously an exception should be raised for the missing character, called as shown in Figure 25, by determining whether the current node belongs to the primary cases, and if not, then determine whether the statement is finished, and if so, throwing that exception.

3) The third is when parentheses appear in the current statement, then the program would assume that a function is currently being called or a variable is being indexed. However, if the other half of the brackets are missing or if there are square brackets on the left and round brackets on the right, an exception will be thrown. This exception occurs in three places, the first being when parsing the primary expression (Figure 26). Since it is not convenient to set up an if statement to determine this here, I have chosen that after determining other types of exceptions, the remaining case must be this type of exception; the second is when parsing an expression containing parentheses (Figure 27), throwing an exception of that type if the right-hand side never finds the right half of the parentheses



```

def parse_primary_expression(self):
    if self.get_token_type() == TokenType.KEYWORD:
        # not parsed as expression,
        # instead, parsed by other statement parser method following expression statement
        return None
    if self.get_token_type() in self.primary_cases:
        return self.primary_cases[self.tokens[0].get_type()]()

    token = self.get_token()
    if self.get_token_type() == TokenType.EO_STMT:
        raise InvalidExpressionError2(token.row, token.col)
    if self.get_token_type() in NON_PREFIX_OPERATOR_TOKENS:
        raise InvalidOperatorError(token.row, token.col)
    raise InvalidExpressionError3(token.row, token.col)

```

Figure 25: Invocation of the "InvalidExpressionError2" exception class

during traversal; the third is parsing an expression containing square brackets (Figure 28), essentially the same principle as the second call.

```

def parse_primary_expression(self):
    if self.get_token_type() == TokenType.KEYWORD:
        # not parsed as expression,
        # instead, parsed by other statement parser method following expression statement
        return None
    if self.get_token_type() in self.primary_cases:
        return self.primary_cases[self.tokens[0].get_type()]()

    token = self.get_token()
    if self.get_token_type() == TokenType.EO_STMT:
        raise InvalidExpressionError2(token.row, token.col)
    if self.get_token_type() in NON_PREFIX_OPERATOR_TOKENS:
        raise InvalidOperatorError(token.row, token.col)
    raise InvalidExpressionError3(token.row, token.col)

```

Figure 26: First call to "InvalidExpressionError3"

```

def parse_paren_expression(self):
    self.tokens.pop(0) # remove left paren
    node = self.parse_logic_or_expression()
    if node and self.get_token_type() == TokenType.R_PAREN:
        self.tokens.pop(0) # remove right paren
    else:
        token = self.get_token()
        raise InvalidExpressionError3(token.row, token.col)
    return node

```

Figure 27: Second call to "InvalidExpressionError3"

Next is the "end" terminator missing exception, with a total of 4 calls. The first, second and third are all cases where "end" is not detected when parsing a selection statement (Figure 29), the difference being that in the first case the MATLAB code only writes if/switch, in the second case it writes additional elseif/case branch, and in the third case it writes an else/otherwise branch, all three of which should throw an exception if "end" is not detected. The fourth case is where the "end" is not detected when parsing the loop statement (Figure 30), and the principle of the call is exactly the same as before.

Finally, there is the exception for incomplete statements encountered when parsing square bracketed expressions (Figure 31). With the judgement that the function encountered a node with the content None without finding the terminator, so clearly the statement is not complete and the exception should be thrown. So far, the only sample error I can think of is the case of parsing square bracketed expressions, and I think there are still some cases that I have not found.

```

def parse_index_list(self):
    root = ASTNode(n_type=ASTNodeType.INDEX_LIST_EXP)
    while self.get_token_type() != TokenType.R_PAREN:
        if self.get_token_type() == TokenType.COLON:
            root.add_child(ASTNode(n_type=ASTNodeType.CLN_EXP, n_text=self.tokens.pop(0).get_text()))
        else:
            child = self.parse_logic_or_expression()
            root.add_child(child)

        if str(self.get_token()) == ",":
            # one argument finished, continue to parse another argument
            self.tokens.pop(0)
        else:
            if self.get_token_type() != TokenType.R_PAREN:
                token = self.get_token()
                raise InvalidExpressionError3(token.row, token.col)
    return root

```

Figure 28: Third call to "InvalidExpressionError3"

```

def parse_selection_statement(self):
    if self.get_token().get_text() not in ("if", 'switch'):
        return None
    node = ASTNode(n_type=ASTNodeType.SEL_STMT, n_text=self.get_token().get_text())

    start_token = self.tokens.pop(0)

    # firstly a if/switch
    clause = self.parse_selection_clause(start_token.get_text())
    if clause is None:
        raise EndMissingError(start_token.row, start_token.col)
    node.add_child(clause)

    a, b = SEL_CLAUSES_MAP[start_token.get_text()]
    # then unlimited elseif/case
    while self.get_token().get_text() == a:
        clause = self.parse_selection_clause(self.tokens.pop(0).get_text())
        if clause is None:
            raise EndMissingError(start_token.row, start_token.col)
        node.add_child(clause)

    # finally sometimes a else/otherwise
    if self.get_token().get_text() == b:
        clause = self.parse_selection_clause(self.tokens.pop(0).get_text())
        if clause is None:
            raise EndMissingError(start_token.row, start_token.col)
        node.add_child(clause)

    self.tokens.pop(0) # remove 'end'

```

Figure 29: Parsing the selection statement to call "EndMissingError"

#### 6.4.3.3 Semantic analysis phase (operations.py, interpreter.py, utils.py)

The first is that it does not apply to unary operations on String, such as "+'str'", which obviously has no positive or negative properties and for which the inverse ' ' operation cannot be performed, as shown in Figure 32, which determines whether an exception is thrown by determining whether the operand is String type.

The second one is not applicable to binary operations concerning String, such as multiplication '\*' and left division and right division operations in matrices (Figure 33, Figure ??). Whenever at least one of the two operations of the binary operation is String type, the corresponding exception is thrown. There are also many binary operations such as subtraction '-', dot product '.', '\*', etc. To save space, these can be found in the 'operations.py' file, and the principle of calling exception classes is the same.

The third one is that the multiplication operation of two matrices throws an exception because the matrix dimension does not match the rules of the operation (Figure 35)

The fourth is a comparison operation concerning String (Figure 36). When a comparison operation is encountered, the operands on both sides are detected and an exception is thrown if one side is not String.

The fifth is the type of exception that is thrown when an undefined function or variable is encountered (Figure 37). By detecting whether the identifier to be retrieved is in the list of variables

```

def parse_iteration_statement(self):
    if self.get_token().get_text() not in ('while', 'for'):
        return None
    node = ASTNode(n_type=ASTNodeType.ITR_STMT, n_text=self.get_token().get_text())
    start_token = self.tokens.pop(0)

    clause = self.parse_iteration_clause(start_token.get_text())
    if clause is None:
        raise EndMissingError(start_token.row, start_token.col)
    node.add_child(clause)

    self.tokens.pop(0) # remove 'end'

    return self.complete_statement(node)

```

Figure 30: Parsing the loop statement to call "EndMissingError"

```

def parse_bracket_expression(self):
    self.tokens.pop(0) # remove left bracket
    node = ASTNode(n_type=ASTNodeType.ARRAY_LIST_EXP)
    while self.get_token_type() != TokenType.R_BRACKET:
        if self.get_token_type() == TokenType.EQ_STMT:
            self.tokens.pop(0) # remove unnecessary delimiters

        if self.get_token_type() is None:
            raise IncompleteStatementError()

        child = self.parse_logic_or_expression()
        node.add_child(child)

        while self.get_token_type() == TokenType.EQ_STMT:
            node.add_child(ASTNode(n_type=ASTNodeType.EQ_STMT, n_text=self.tokens.pop(0).get_text()))
    self.tokens.pop(0) # remove right bracket
    return node

```

Figure 31: Invocation of "IncompleteStatementError"

or keywords already defined, and throwing an exception if it is not.

The last one concerns the concatenation operation of matrices (Figure 38), which is divided into two types, one for horizontal concatenation and one for vertical concatenation. When the two matrices need to be concatenated in different dimensions, an exception is thrown for dimensional inconsistency.

#### 6.4.4 Test work for error handling (test\_error\_handling.py)

The testing process of the project is implemented with the help of Python's unit testing function. In the unit test file, the program will first run the MATLAB command (The path to the matlab program needs to be added to the environment variable) "matlab -nosplash -nodesktop -nojvm -batch run('path')logfile temp" through the terminal. It can run the MATLAB scripts I set up by not launching the MATLAB GUI. The output is then saved to a txt file (e.g. for the lexical analysis file "test\_1\_invalid\_character\_error.m", the MATLAB output is shown in Figure 39). It will then run our python project program to save the output to another txt file (Figure 40). The MATLAB output contains unnecessary content, i.e. the parts marked in blue in Figure 26, which can be filtered out using regular expressions. Finally, we can conclude whether the output of the MATLAB and python programs are equal, for example by testing the lexical analysis run as shown in Figure??.

#### 6.4.5 Error Handling Conclusion

For the error handling section, there still need to add more exception types, as there is a limit to what one person or group can do, and inevitably there will be cases of omissions.

```

def parse_bracket_expression(self):
    self.tokens.pop(0) # remove left bracket
    node = ASTNode(n_type=ASTNodeType.ARRAY_LIST_EXP)
    while self.get_token_type() != TokenType.R_BRACKET:
        if self.get_token_type() == TokenType.EO_STMT:
            self.tokens.pop(0) # remove unnecessary delimiters

        if self.get_token_type() is None:
            raise IncompleteStatementError()

        child = self.parse_logic_or_expression()
        node.add_child(child)

        while self.get_token_type() == TokenType.EO_STMT:
            node.add_child(ASTNode(n_type=ASTNodeType.EO_STMT, n_text=self.tokens.pop(0).get_text()))
    self.tokens.pop(0) # remove right bracket
    return node

```

Figure 32: Parsing the unary operation call "UnaryOperatorForStrError"

```

# binary
def evaluate_matrix_multiplication_operation(operand_0, operand_1):
    if isinstance(operand_0, String) or isinstance(operand_1, String):
        raise OperatorForStrError('*')

```

Figure 33: Invocation of "OperatorForStrError"

```

def evaluate_matrix_right_division_operation(operand_0, operand_1):
    if isinstance(operand_0, String) or isinstance(operand_1, String):
        raise ErrorUsingDivision('/')
    if operand_0.size == (1, 1) and operand_1.size == (1, 1):
        return evaluate_array_right_division_operation(operand_0, operand_1)
    # return evaluate_matrix_multiplication_operation(operand_0, inv_matrix(operand_1))

def evaluate_matrix_left_division_operation(operand_0, operand_1):
    if isinstance(operand_0, String) or isinstance(operand_1, String):
        raise ErrorUsingDivision('\')
    if operand_0.size == (1, 1) and operand_1.size == (1, 1):
        return evaluate_array_left_division_operation(operand_0, operand_1)

```

Figure 34: Invocation of "ErrorUsingDivision"

```

# binary
def evaluate_matrix_multiplication_operation(operand_0, operand_1):
    if isinstance(operand_0, String) or isinstance(operand_1, String):
        raise OperatorForStrError('*')

    if operand_0.size == (1, 1) or operand_1.size == (1, 1):
        compat(operand_0, operand_1)
        return evaluate_array_multiplication_operation(operand_0, operand_1)
    if operand_0.size[1] == operand_1.size[0]:
        m = operand_0.size[0]
        n = operand_1.size[1]
        k = operand_0.size[1]
        result = []
        for row in operand_0.rows():
            for col in operand_1.cols():
                num = 0
                for i in range(k):
                    num += row[i] * col[i]
                result.append(num)
        return Double(result, size=(m, n))
    else:
        raise ErrorUsingMultiply()

```

Figure 35: Invocation of "ErrorUsingMultiply"

```
def evaluate_relational_operations(operand_0, operand_1, operator):
    if isinstance(operand_0, String) != isinstance(operand_1, String):
        # one is String while one is not String
        raise CompareWithStrError(operator, operand_0, operand_1)
    fun = {
```

Figure 36: Invocation of "CompareWithStrError"

```
def retrieve(self, identifier):
    if identifier in self.variables:
        return self.variables[identifier]
    if identifier in self.builtins:
        return self.builtins[identifier]
    raise UnrecognizedFunctionOrVar(identifier)
```

Figure 37: Invocation of "UnrecognizedFunctionOrVar "

```
def concatenate(data_list, direction):
    if direction == "horz":
        if len(set([data.m for data in data_list])) > 1:
            raise DimensionNotConsistentError('horzcat')
        data = sum((sum(list(tup), []) for tup in zip(*[data.rows() for data in data_list])), [])
        size = (data_list[0].m, sum(data.n for data in data_list))
        cls = find_nearest_common_ancestor(set(data.get_class() for data in data_list))
        return cls(data, size)
    if direction == "vert":
        if len(set([data.n for data in data_list])) > 1:
            raise DimensionNotConsistentError('vertcat')
        data = sum((sum(data.rows() for data in data_list), [], [])
        size = (sum(data.m for data in data_list), data_list[0].n)
        cls = find_nearest_common_ancestor(set(data.get_class() for data in data_list))
        return cls(data, size)
```

Figure 38: Invocation of "DimensionNotConsistentError"

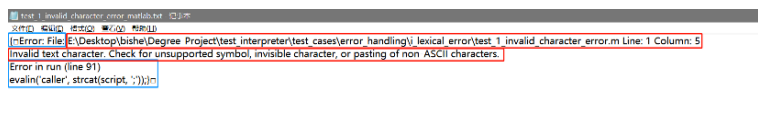


Figure 39: matlab output

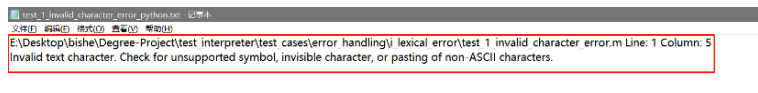


Figure 40: Figure 27 Python output

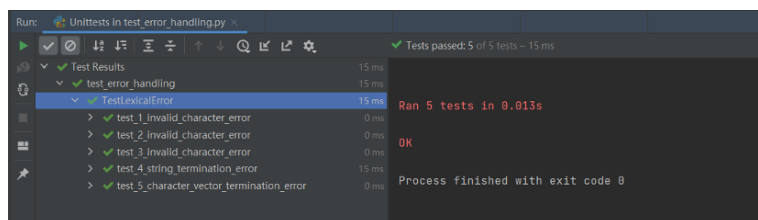


Figure 41: Figure 28 Results of the lexical analysis unit test in Pycharm

## 7 Hao Tingting's Individual Part

This part mainly describes the test part of the compiler. Implement automated testing capabilities, take the framework for Unittest and develop scripts to generate random test programs to test features developed by other team members. In addition, when we found that the behavior of our software was not consistent with what was defined in the documentation, we communicated with other team members promptly. We resolved some output errors, such as Solve the problem of symbols and operators in arrays. This part of the work is done by HAO TINGTING, whose UCD student number is 17205878.

### 7.1 Declaration of Authorship

I declare that all of the following are true:

- 1) I fully understand the definition of plagiarism.
- 2) I have not plagiarized any part of this project and it is my original work.
- 3) All material in this report is my/our own work except where there is clear acknowledgement and appropriate reference to the work of others.

### 7.2 Preparation

#### 7.2.1 The importance of testing in our project

The software running on the computer is processed by a compiler or compiler-like tool, and the compiler is an important part of building software infrastructure, so its correctness is critical[7]. Running the debug program on the wrong compiler may cause unexpected behavior, leading to software failures, such as complex bugs in the Java 7 implementation that have caused several major Apache projects to crash[7]. Compiler errors can have serious unintended consequences and make a software debugging very difficult. For application developers, this error is difficult to detect and discover because it is difficult to determine whether the software failure is caused by the program being developed or by the compiler being used[5]. Therefore, the reliability of the compiler is critical. Moreover, systematic testing and quality assurance for compilers are important. In this project, the testing part of the work is also indispensable.

#### 7.2.2 Why we use MATLAB for comparison

In our project, a challenge is the complexity of the compiler's input and output language representations. We needed a canonical interpreter to explain what our project was supposed to do or what the output was. When testing the compiler, we need at least one well-designed compiler to use as our comparison. Then, compare the results from compilers to see if a compiler bug has been detected. We learned that Hawblitzel et al. [17] proposed several other compiler differences tests to select the implementation to compare.

- Cross-compilation strategy: Detects compiler errors by comparing the results compiled by different compilers. This policy is one of the most common in compiler difference testing [7].
- Cross-optimization strategy: Compiler errors are detected by comparing different levels of optimizations implemented in the compiler. This policy has been widely applied in compiler testing research[7].
- Cross-version strategy: Compiler errors are detected by comparing the results of different versions of a single compiler [7].

So, to check that our results are correct, we take the Cross-compilation strategy. We need a reference interpreter, and here we choose MATLAB, which is a very powerful software package with many built-in tools to solve problems and develop graphic illustrations [12]. Moreover, MATLAB systems are technically known as interpreters, and MATLAB is an interpretive language. It means that every statement that appears on the command line is translated (interpreted) into a language

that the computer can better understand and then executed immediately [14]. It shows that in the test section, we compare the running results with the results of MATLAB.

### 7.2.3 Something about testing

Compiler testing generally consists of "Standard syntax testing + Functional testing + Optimization testing + Random testing" and other methods. Compiler optimization technology is the core of programming language and language virtual machines, so the testing method can also be mostly reused to test programming language and language virtual machines[4]. The functional test of any object generally starts from the analysis of input and output. The compiler's input is a certain language, and each language has its syntax and semantics specification. The main part of the compiler test is to test the syntax meaning specification, which is called the standard test. Standard testing mainly includes type definition, type conversion, operator operation.

In this section, to ensure compiler behavior, the basic functions and output are correct, we mainly implemented functional testing.

## 7.3 Description of my work

### 7.3.1 Manual testing

We started with manual testing. In short, it is to use examples to manually input different transform operation symbols and transform operands to test and verify our interpreter. This section mainly examines the basic output, including addition, subtraction, multiplication, and division of decimals, simple for and if and while statements. Such a test program can be effective in finding bugs because the test code can be adjusted for specific requirements and is better suited to specifically testing the newly implemented functionality.

However, we found some serious problems (listed below), which took me much time to correct. During this time, I need to fix the generated bugs based on HAN's work, which indicates that I need to complement the previous work. For example, HAN implements integer input and addition, subtraction, multiplication, and division operations. Based on his work, I need to improve the operation of floating-point number type and realize the input and output of scientific notation and the operation of the floating-point number.

### 7.3.2 Problems identified and solved

Here are some of the major errors that have been corrected during testing

#### 7.3.2.1 Solve the problem of symbols and operators in arrays

In the test of array representation and simple addition and subtraction, we found a major error. There are positive and negative numbers in the array, and the operation symbol is not clear. Before this problem was found, we treated "-" as operators, and the symbol would be used as a minus sign in the test, which would lead to the problem of not representing negative numbers in the array. Therefore, in this part, we need to consider the influence of Spaces on the algorithm.

My solution: when " " is detected, as figure 42 shows, it means that we are going through the following loop: First, we need to think about whether every number has a sign. If there are only Spaces and no symbols between two numbers, we need to change the space before the following number to "+"; Second, we need to understand that if there is a space before the "+" or "-", we need to change the space before the "+" and "-" to ", ". Then delete the new ", " if the "+" and "-" are preceded by commas and followed by Spaces. Furthermore, at last, please clear all the Spaces before the compiler executes this array.



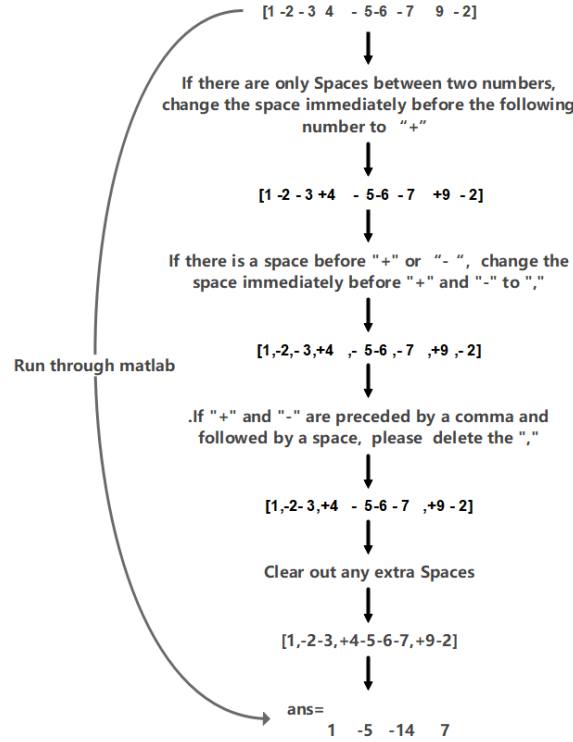


Figure 42: Resolves confusion between symbols and operators in arrays

### 7.3.2.2 Diverse floating-point representation

We found a problem when detecting floating-point operations. That is, after a simple calculation, our output results and MATLAB output has a huge difference. The output of MATLAB is varied, sometimes expressed in a scientific notation, sometimes reserved to four decimal places. Here, we realize that we need to consider implementing the inputs and outputs of scientific notation.

My solution: based on HAN's code framework, we need to redefine a variable as input. Moreover, add a type of variable name to (token\_types.py) and ASTNodeType (node\_types.py), but we also need to add explanations to Parse and Interpreter. In this way, floating-point numbers and scientific notation are identified, respectively

According to my solution, the regular expression looks like this:

```

1 NUMBER_LIT = re.compile(r"[0-9]+\.[0-9]+|[0-9]+\.[0-9]+\.[0-9]+|[0-9]+\.[0-9]+") %double
2 SCIENTIFIC_E = re.compile(r"[0-9]+\.[0-9]*e[+-]?[0-9]+") %scientific notation

```

However, to make the code cleaner, we added the scientific notation representation directly into Number\_Lit. When the regular expression detects pattern matching and replacement, the scientific notation also matches the original.

After optimizing the regular expression looks like this:

```

1 "NUMBER = r"[0-9]+\.[0-9]+|[0-9]+\.[0-9]+\.[0-9]+|[0-9]+\.[0-9]+ "
2 NUMBER_LIT = re.compile(rf"({NUMBER})[eE][+-]?[0-9]+|{NUMBER}")

```

Under the circumstance of realizing the input of scientific notation, we spent more time in MATLAB to discover the output law of scientific notation. During this period, my work was to find rules. We tried a lot of critical conditions to understand the rules of the output results of MATLAB.

I found that when the result of the calculation:

1. Abs(integer) greater than or equal to 10<sup>9</sup>.
2. The absolute value of a decimal is greater than or equal to 1000.
3. The absolute value of a decimal is 0.001 or less.



In all three cases, the output in scientific notation. In other cases, the output is a floating-point number (leaving four decimal places behind the decimal point).

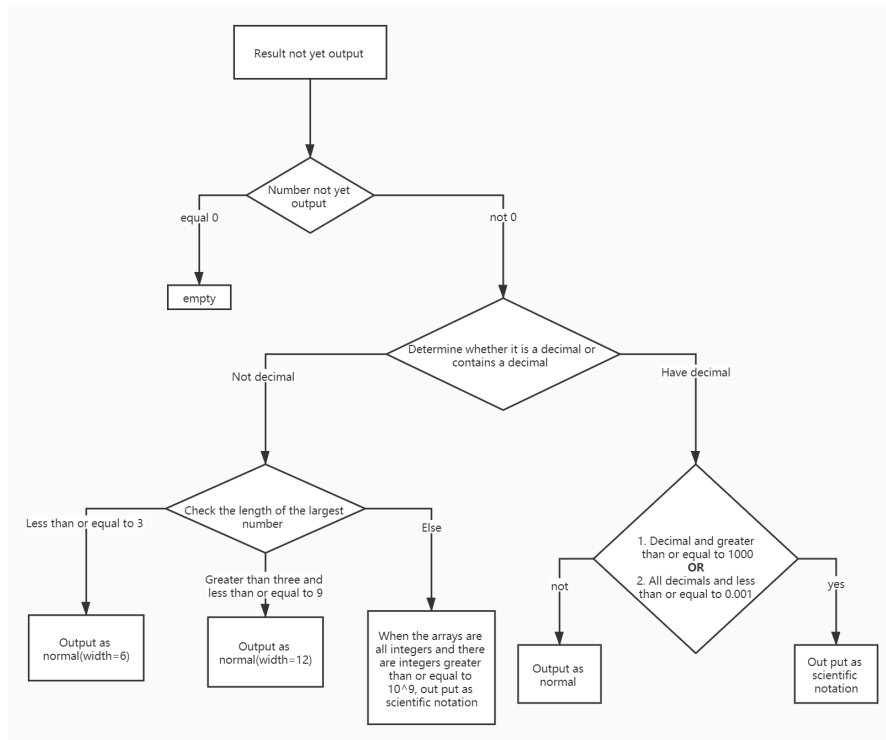


Figure 43: Summarized rules concerning the representation of output results

Figure 43 is another way I tried to express the rules I summarized about representing in Scientific notation. We will encounter operations of type array and double. Based on our code design, we will let the compiled result determine the representation of the output based on the design shown in the figure.

### 7.3.2.3 Solve matrix addition and subtraction operation and representation problems

I added matrix addition and subtraction to the array. The representation of the output is also complex. I still need to spend time looking for rules in MATLAB. At this time, I found that some special rules, such as sometimes the expression of matrix calculation, results sometimes need to put forward scientific notation.

Here's my summary of the rules for using scientific notation in arrays:

1. if the array is full of integers and if any of the integers are greater than or equal to  $10^9$ , when represented the result, the scientific notation is pulled out of the array.
2. In the presence of decimal numbers, any number greater than or equal to 1000, or all numbers less than 0.001, will cause the output to be represented in scientific notation.

When the value of the result does not meet the rules as mentioned above, the output does not require scientific notation but still follows the floating-point representation rules.

### 7.3.3 Automatic test

#### 7.3.3.1 Automatically compare test results

Although the test whose code through typing helped us find the error, and our calculation results at the present stage were basically consistent with those of MATLAB, it was still not intuitive enough and some details might be ignored, such as the specific bit width of the results may be different.

After some investigation, we discovered that technicians had developed different test techniques for automated test compilers. After some research, we learned about some Python testing frameworks [6].

a) UnittestPython's default test automation framework makes it easier to run a single test case on the smallest module of a program [19]. Unittest provides assertions such as `assertEqual`, `assertLn`, `assertTrue`. However, it supports so many abstract methods that the purpose of the test code is sometimes unclear.

b) Robot Framework: Mainly used for test-driven development and acceptance. It simplifies the overall automation process by making it easy for testers to create readable test cases using a catch-word-drive-test approach. However, the Robot Framework is cumbersome in creating custom HTML reports. It also has no advantage in parallel testing [10].

c) Pytest: Development teams and various open-source projects often use the tool because it is open source. Pytest uses assert expressions directly, and the native Python assertion functions allow users to write more compact test suites. While it makes it easy for users to write test cases, it cannot be used with any other test framework[13].

After learning about these frameworks and based on our project requirements, we adopted unit test and organized the test cases together in a class manner. Fortunately, PyCharm provides flexible running and debugging support for test scripts. In order to test all the scripts in one test runner script, we firstly create some classes that are inherited from the `unittest.TestCase` class for every directory, in which scripts that test the same category of features are stored together. In each class, we encapsulate the test methods that implement the `assertEqual` function from the unittest frame. The test class will pass the unit test if and only if all of the `assertEqual` methods implement in it passes the test. Moreover, the runner script will pass the unit test if and only if all of its test classes pass the test. Therefore, it is convenient to see if any script failed the test or all of the cases passed. Figure44 shows that the results of our compiler are not the same as those of MATLAB and indicate errors, and Figure45 shows that our compiler passes all tests.

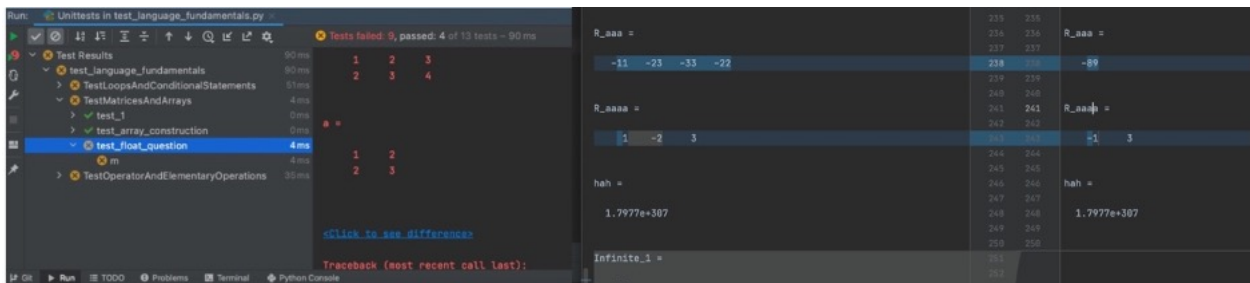


Figure 44: Show the difference between MATLAB(left part) and miniMATLAB(right part)

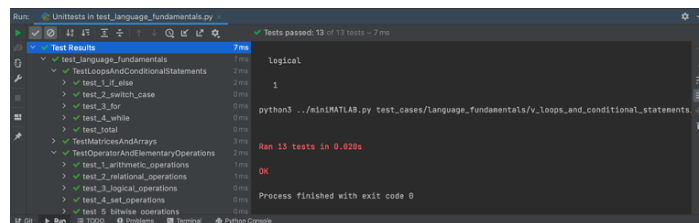


Figure 45: Pass the test

### 7.3.3.2 Randomly generate test code:stage 1 and stage 2

While manually constructed test programs can be effective in finding bugs, it is not easy to build

effective, diverse test programs that meet specific test requirements. In this part, our idea is to randomly combine all kinds of grammars of the language, including variable names, expressions, operators, on the premise of ensuring the accuracy of the grammar, to generate a test code that is controllable in code length, code nesting depth, number of arguments.

#### Stage 1:

We realized the automatic test that the expression structure is fixed. At this stage, we used Python's built-in function (random module) to generate floating-point numbers within the specified range randomly. However, the operating symbols at this stage still need to be set manually. Therefore, the mode of automated testing at this stage is very fixed and still needs to be set up manually. However, compared to the previous manual tests, this test expanded the test sample. Make the test operands more random in case something happens that we have not tested before.

#### Stage 2:

While the initial automated test generator is effective, it still needs to be set up manually. We want to implement a higher level of the random generator to generate expressions with random structure. In this generator, operands and generate operators can randomly be generated, and we can also randomly generate case times and test some loops and conditional statements.

We learned that as an algorithm, recursion is widely used in programming languages. It usually refers to using methods of the function itself in the function definition, which is equivalent to a depth-first walk of the tree [24]. In this way, a large and complex problem is usually broken down into a small one which similar to the original problem. A recursive strategy would certainly reduce the amount of program code, since only a few procedures would be used to describe solving the duplicated computation problem. The power of recursion lies in the finite number of statements that can define an infinite set of objects [21]. After recursing, people get a slightly more complicated loop, but as long as it is not an infinite loop, people can iterate over it as many times as people want to get the result.

So, using this idea, we need to set up the recursive forward segment, recursive return segment and set up the boundary conditions[23]. When the boundary conditions are not met, recursive processing is required. When the boundary conditions are met, the recursion returns; thus, objects can be arbitrarily combined indefinitely. In this case, recursive progression refers to when the number of recursive levels is less than the maximum number of recursive levels we set; at this point, we continue to generate operators or operands in this part randomly. When the maximum number of recursive levels is reached, we need to terminate recursion to avoid an endless loop.

This testing approach expands the case of the test sample and covers tests that implement more complex scenarios. We can also optimize the tests for stage1 and change the number of recursion levels we want at will. The generated code for testing the arithmetic operation is shown here, as shown in Figure46. The figure on the left shows our test code in Stage 1, where the symbols in the middle remain unchanged and the operands on the left and right sides are randomly generated. During the stage 2 phase, we can set the recursion times to randomly generate operands and operand symbols in the same arithmetic expression. This section describes recursion at the expression level because it uses a recursive approach that allows different operators to appear in an expression. The logical operators, relational operators, arithmetic operators of the current stage may simultaneously appear in the same expression. The test at this point implements the option combination test, which randomly combines the options and then tests some specific code.

After ensuring that our calculation results are consistent with MATLAB, we can also test more flexible cases, such as if and while loops detect complex cases. For example, in an expression such as  $M+N$ , the operand  $M$  can be either a terminator like an identifier or number or another expression consisting of another operator and one or more operands. In this way, we realized that expressions are recursive and statements are recursive.

$$\text{RandomExpression} \Rightarrow \begin{cases} \text{RandomBinaryExpression,} & \text{case1} \\ \text{RandomPrefixExpression,} & \text{case2} \\ \text{RandomPostfixExpression,} & \text{case3} \\ \text{Terminator,} & \text{case4} \end{cases}$$

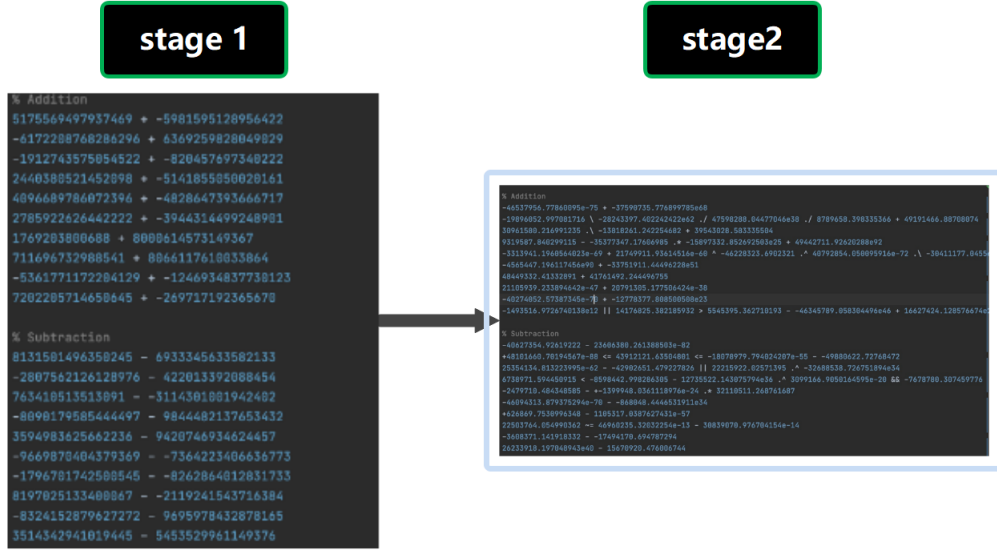


Figure 46: Randomly generated code

RandomBinaryExpression  $\Rightarrow$

RandomExpression+RandomBinaryOperator+RandomExpression

RandomPrefixExpression  $\Rightarrow$  RandomPrefixOperator + RandomExpression

RandomPostfixExpression  $\Rightarrow$  RandomExpression + RandomPostfixOperator

After such a test, we also found that there were some problems with our operations. Before the test, we did not consider the priority and complexity of the compiler's operations. When there are many operation symbols simultaneously, our compiler cannot correctly judge the priority of the operation, which shows that this kind of option combination test is necessary.

## 7.4 Results

After repeated modifications and adjustments, we find most of the rules of MATLAB output results. Besides, we solve the diversity of floating-point arithmetic results, solve the array's operating symbol and operator problems in the array, and solve the array addition and subtraction arithmetic and representation problems. We can ensure that the output results of mini-MATLAB and MATLAB results maintain a high degree of consistency.

In addition, in this section, we upgrade manual tests to automated tests. In this part, we have implemented a test generator that automatically compares the compiler results with MATLAB results and has implemented a test generator that automatically randomly generates many test combinations. At this time, we can freely expand the range of testing and complexity of detection, and the input and output are also consistent with MATLAB. The errors in the tests and the random generator implementation proved that our compiler had passed the tests perfectly.

## 7.5 Discussion and future work

There are thousands of tests in our group, most of which are validation tests, such as checking if a function is correct, verifying the result of a calculation, or the number of loops. In this project, when we did our tests, our tests focused more on finding and solving problems that we had not thought of before or not having the generated code get the wrong results. If we have more time, we would like to do something else in this part:

1. Do Special test: Performance test can detect whether the compiler's performance has reached the established target, mainly including execution performance and code volume. Constructing an invalid program is of limited use to test the compiler because a program has to go through multiple processing stages by the compiler. If the compiler provides an invalid input program, it is often discarded during the initial phase of processing [7].

2. Realize the left division and right division of the matrix by calling the methods we implemented. In addition, the matrix aspect of the test is also worth considering seriously. MATLAB is convenient for matrix arithmetic, since it can use the matrix as a basic storage unit, so it is important to do this task. Besides, do the division work using a function from the Numerical Python(inv) is also a good idea.

3. Compiler test efficiency: Although we have improved the efficiency of compiler testing to some extent, compiler testing is still a time-consuming and trivial task. We need to make further efforts to improve compiler tests' efficiency, make compiler tests more efficient, and generate test optimizations. The basic idea is to build an optimization model that the compiler executes and then generate tests that include optimization opportunities to test whether the parameters passed to the function are received as they are.

4. In compiler testing, errors are often found many times by the testing program. Not only does this lead to efficiency issues, but it also creates much extra work for developers to review and classify duplicate test programs. We want to design a test program that generates only new bugs or elicits feedback from the test program using known bugs.

## 7.6 Conclusion

Compiler developers attaches significant importance to testing because it reveals the robustness and consistency of the compiler system. We have achieved not only the diverse representation of floating-point numbers and arrays, but also the automatic comparison between the results of our compiler and that of MATLAB. Moreover, our automatic generator can generate test code which contains random expressions including different operators and various operating conditions. After the continuous improvement to the compiler, we maintained the consistency of compiling results. We have finally passed the automatic test and obtained a relatively perfect test code. Despite that we have successfully implemented basic functional testing in terms of compiler testing, we still have high-level plans including optimizing the test code to be more efficient and effective in the future.

## 8 Overall Discussion and Conclusions

Comparing to official MATLAB software, our project have some limitation on these aspects:

### 8.1 The limitation on functionalities

Although we have realized a few functionalities of MATLAB, and carried out different kinds of testing, comparing to the official MATLAB software, the complexity of our project is still a tip of the iceberg.

We realized four different data types, however there are a number of other data types. It's easy to infer that with more data types the type system would become much more complex.

Our data type support up to two dimensional array, but actually in the official software, the dimension of array is not limited. Since it involves the matrix operation as well as the output format, this is also considered as a very difficult task for us.

MATLAB is most powerful in its built-in functions. These functions are very helpful in scientific calculations. However, realized few of them.

### 8.2 The limitation on error reporting

For a professional interpreter or compiler, the error reporting should ensure that no matter what the input looks like, the language processor will never crash, and it could always report a proper error. This is what we have not realized, and a direction to make effort on. In other words, improve the robustness of the software.

### 8.3 The limitation on performance

MATLAB use C for its core, while we used python. Therefore, it is obvious that the speed of our software is not comparable to MATLAB from the implementation basis view, not to mention the optimizing in memory and algorithm level. To improve the speed, it might be a good idea to apply some package for scientific calculation in python, such as numpy.

## 9 Suggestions for possible future work.

As mentioned in the previous section, on our project there are some point that worth continuous researching.

1. We have some unsolved bugs
2. There are some desired upgrade, includes:
  - support complex number
  - support self-defined function
  - support object oriented programming
  - support higher dimensional array
  - support more built-in functions
3. There are also two main direction to optimize:
  - the speed of running, as mentioned in the previous section
  - the robustness of error reporting, as mentioned in the previous section.
4. The knowledge in compiling
  - Although we found it is difficult to develop the back end, it does not influence the interest of us. Completing this part is just a matter of time.

## References

- [1] Joël Akeret, Lukas Gamper, Adam Amara, and Alexandre Refregier. Hope: A python just-in-time compiler for astrophysical computations. Astronomy and Computing, 10:1–8, 2015.
- [2] Frances E. Allen and John Cocke. A program data flow analysis procedure. Communications of the ACM, 19(3):137, 1976.
- [3] Frances E. Allen and John Cocke. A program data flow analysis procedure. Communications of the ACM, 19(3):137, 1976.
- [4] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. ACM SIGPLAN Notices, 24(7):146–160, 1989.
- [5] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In Proceedings of the 38th International Conference on Software Engineering, pages 180–190, 2016.
- [6] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In Proceedings of the 38th International Conference on Software Engineering, pages 180–190, 2016.
- [7] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. ACM Computing Surveys (CSUR), 53(1):1–36, 2020.
- [8] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. Mathematics of computation, 19(90):297–301, 1965.
- [9] Antonio Cuni. High performance implementation of Python for CLI/.NET with JIT compiler generation for dynamic languages. PhD thesis, PhD thesis, Dipartimento di Informatica e Scienze dell’Informazione . . . , 2010.
- [10] Alexandre da Silva Simões, Esther Luna Colombini, Jackson Paul Matsuura, and Marcelo Nicoletti Franchin. Torp: The open robot project. Journal of Intelligent & Robotic Systems, 66(1):3–22, 2012.
- [11] Miles Ellis. Is there a role for standards in the future of fortran? In ACM SIGPLAN Fortran Forum, volume 17, page 5. ACM New York, NY, USA, 1998.
- [12] Delores M Etter, David C Kuncicky, and Douglas W Hull. Introduction to MATLAB. Prentice Hall, 2002.
- [13] Hans Fangohr, Vidar Fauske, Thomas Kluyver, Maximilian Albert, Oliver Laslett, David Cortés-Ortuño, Marijan Beg, and Min Ragan-Kelly. Testing with jupyter notebooks: Notebook validation (nbval) plug-in for pytest. arXiv preprint arXiv:2001.04808, 2020.
- [14] Amos Gilat. MATLAB: An introduction with applications, volume 3. Wiley New York, 2008.
- [15] Zhi Guo, Betul Buyukkurt, John Cortes, Abhishek Mitra, and Walild Najjar. A compiler intermediate representation for reconfigurable fabrics. International Journal of Parallel Programming, 36(5):493–520, 2008.
- [16] Mary Hall, David Padua, and Keshav Pingali. Compiler research: the next 50 years. Communications of the ACM, 52(2):60–67, 2009.

- [17] Chris Hawblitzel, Shuvendu K Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. Will you still compile me tomorrow? static cross-version compiler validation. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pages 191–201, 2013.
- [18] Manuel Hohenauer and Rainer Leupers. C Compilers for ASIPs. Springer, 2010.
- [19] Christian Hovy and Julian Kunkel. Towards automatic and flexible unit test generation for legacy hpc code. In 2016 Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE), pages 1–8. IEEE, 2016.
- [20] Steve Keckler. Proceedings of the 36th Annual International Symposium on Computer Architecture. Association for Computing Machinery, 2009.
- [21] Wolfgang Kunz and Dhiraj K Pradhan. Recursive learning: a new implication technique for efficient solutions to cad problems-test, verification, and optimization. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 13(9):1143–1158, 1994.
- [22] Bruce W Leverett, Roderic G Cattell, Steven O Hobbs, Joseph M Newcomer, and Andrew H Reiner. An overview of the production quality compiler-compiler project. 1979.
- [23] Minh-Thang Luong, Richard Socher, and Christopher D Manning. Better word representations with recursive neural networks for morphology. In Proceedings of the seventeenth conference on computational natural language learning, pages 104–113, 2013.
- [24] Albert Marcet and Ramon Marimon. Recursive contracts. 2011.
- [25] Michael F Morgan. The cathedral and the bizarre: An examination of the viral aspects of the gpl. J. Marshall J. Computer & Info. L., 27:349, 2009.
- [26] KR Srinath. Python–the fastest growing programming language. International Research Journal of Engineering and Technology, 4(12):354–357.
- [27] YUAN Yan. Design and realization of compiler principle web based learning system [j]. Journal of Chengde Petroleum College, 1, 2006.