CS4ZP6 Capstone Group 4 Project Documentation

Lexical, Syntax and Semantic Modules for Pangu Compiler in Python

Supervisor: Prof. Franek

April 2020

Name	**Email**
Yujing Chen	cheny48@mcmaster.ca
Ziqing(Amy) Xu	xuz69@mcmaster.ca
Baikai Wang	wangb40@mcmaster.ca
Wei Jiang	jiangw41@mcmaster.ca
Quan(Steve) Man	manqz@mcmaster.ca

Table of Contents

1. Description

The proposed system is a Python based compiler for Pangu. Pangu is an imperative programming language similar to Java or C++ designed at McMaster University by professors Franek and Liut to manage memory through the machinery of primary and secondary references. The compiler is proposed to be a stand-alone program used in Linux environment as a command line invocation. The compiler will translate the Pangu source code to the target code PIC which is in fact a lower level intermediate code. First, a parser generated by PLY (Python Lex-Yacc) will translate Pangu source code into an abstract syntax tree. Then the abstract syntax tree will be transformed in several steps into the target code PIC. The PIC file will be interpreted by an interpreter which is not a part of this project and will be facilitated by our supervisor Prof.Franek. At this stage, we have accomplished the lexical, syntax and semantic modules for the Pangu compiler and code generator module will be implemented in further steps.

2. Software Requirement Specification

[Steve is responsible for section 2.1 - 3.1;
Wei is responsible for section 3.2 - 3.5;
Amy is responsible for section 4.1 - 4.3;
Yujing is responsible for section 4.4;
Baikai is responsible for section 4.5]

2.1 Purpose

This document will serve to document and formalize the requirements of the Pangu interpreter.

2.2 Definitions

Word/Phrase	Definition
Abstract Syntax Tree	abstract syntactic structure of source code
Pangu	new language developed at McMaster

	which handles emphasizes on handling memory allocation/deallocation using primary references
PIC	lower level intermediate code that is then translated into machine code
Symbol table	a data structure where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source

2.3 Background

This documentation is only for the lexical, syntax and semantic modules of the Pangu compiler.

2.4 References

[1] "sys - System-specific parameters and functions¶," *sys - System-specific parameters and functions - Python 3.8.2 documentation*. [Online]. Available: https://docs.python.org/3/library/sys.html. [Accessed: 26-Apr-2020].

[2] *PLY (Python Lex-Yacc)*. [Online]. Available: https://www.dabeaz.com/ply/ply.html. [Accessed: 26-Apr-2020].

[3]830-1998 — IEEE Recommended Practice for Software Requirements Specifications. 1998. doi:10.1109/IEEESTD.1998.88286. ISBN 978-0-7381-0332-7

[4]F. Franek and M. Liut, *Designing and implementing a compiler*. Unpublished manuscript. Hamilton, Ontario, 2019.

3. Overall Description

3.1 Product Perspective

There are 3 main modules: pangulex, panguyacc and panguseman. Pangulex is the scanner/tokenizer and passes tokens to panguyacc which parses the tokens into an Abstract Syntax Tree. Then finally, panguseman will perform semantic checks with the help of pangusymtab which contains the symbol table. The symbol table is created and maintained by the interpreter.

Users can interact with the program via Command Line by entering “python3 panguyacc.py name_of_test_file.asc”. The parser will then output the parsed results of the selected file if successful or returns some errors.
If the parsing was successful, an abstract data tree of the Pangu source code will be produced. Then the abstract syntax tree will be transformed in several steps including some optimization into PIC. The PIC file will be interpreted (executed) by an interpreter which is not a part of this project.

3.2 Design Constraints

3.2.1 Operations

The program should be able to take a Pangu program in a .asc file and perform lexical, syntax and semantic analysis on it within a reasonable time period. It should detect and report any illegal characters, syntactical and/or semantic errors in the source program. If there is no error, the program should show all tokens/productions, build/show an abstract syntax tree, and/or build/show a symbol table if the user chooses to do so by setting corresponding flags.

3.2.2 Site Adaptation Requirements

The program should be system independent and it should run correctly in a terminal or Windows command prompt.

3.3 Product Functions

3.3.1 Lexical Analysis

Given an input Pangu program, the pangulex.py module can break the input program into a collection of tokens defined by a set of regular expression rules, which can be used by other modules. The generated tokens can be displayed by setting *pangulex.y_show_tokens* flag in *panguyacc.py* module to *True*. Comments can be shown by setting the *y_pangulex.y_show_comments* flag in *panguyacc.py* module to

True. It can also detect and report any illegal characters that are not specified by the set of regular expression rules.

3.3.2 Syntax and Grammar Validation

The program performs syntax and grammar validation on the input Pangu program based on predefined BNF grammars. Syntax errors can be detected and reported by the program. If there is no error, all the production rules involved in the input Pangu program can be shown by setting the *y_show_productions* flag in *panguyacc.py* module to *True*.

3.3.3 Abstract Syntax Tree Construction

An abstract syntax tree of the input Pangu program can be constructed and displayed. The abstract syntax tree provides the abstract syntactic structure of the input program. It can also be used to generate a symbol table.

3.3.4 Symbol Table Generation

A symbol table keeps track of declared identifiers and their various attributes such as scope, kind, and so on. It is used in semantic analysis to make sure all the identifier names used in the input program are uniquely declared before use and they are correctly used in terms of scope and type.

3.4 User Characteristics

Users of the program are those who perform lexical, syntactical and/or semantic analysis on a Pangu program and/or those who continue to build a compiler for Pangu. They are expected to be familiar with Pangu and Python3 so that they can use this program based on their needs by modifying the flags in *panguyacc.py*. For example, if they only need to see the abstract syntax tree, they can set the flags *y_build_ptree* and *y_show_ptree* to *True* and others to *False.*

3.5 Constraints, Assumptions and Dependencies

The program is designed to run in a terminal or Windows command prompt. It requires Python 3.8 or later and a Python PLY-3.11 package to be installed. The input file with .asc extension has to be placed in the same directory as this program.

4. Specific Requirement

4.1 External Interface Requirements

This section provides information of the requirements needed to ensure that the system will communicate properly with external components. Two main required external interfaces are user interface and software interface. Firstly, in our project, the command line interface is chosen as the user interface. Users can communicate through using codes in a command facilitated by the CLI (such as terminal). More details about the command line interface will be discussed in Interface Viewpoint in the Software Design section. Also, the syntax of all the commands are known to the users which will be discussed in the Code Guide section. Then, let’s discuss the software interface requirement. The compiler is proposed to be a stand-alone program used in the Linux environment using a command line invocation. Then, PLY (Python Lex-Yacc) which is a parsing tool is needed for implementing lexical and syntax modules of the compiler. Thus, CLI, Linux OS and PLY are the two main external interface requirements in this project.

4.2 Functional Requirements

In this section, general functional requirements will be discussed. And functional requirements will be more specifically discussed in section 3.5. The main and intuitive function of the compiler is to translate the Pangu source code into the target code PIC which is in fact a lower level intermediate code. First, a parser generated by PLY (Python Lex-Yacc) will translate Pangu source code into an abstract syntax tree. Then the abstract syntax tree will be transformed in several steps including some optimization into PIC. The PIC file will be interpreted (executed) by an interpreter which is not a part of this project and will be facilitated by our supervisor Prof. Franek. At this stage, unfortunately, we have only finished lexical, syntax and semantic modules of the Pangu compiler but have not accomplished type checking and the code generator. Implementing type checking, code generator and some optimizations are the further steps for this project.

4.3 Performance Requirements

In this section, some performance requirements will be discussed. In this project, the performance of intelligent error messaging of the compiler takes precedence over the speed performance, however one-pass approach should be maintained. For example, the compiler will stop and return an error message if there is a syntax or discernible semantic error in the source code. Therefore, at this stage, no error recovery ability will be built into the compiler. Further, some optimizations (such as abstract syntax tree, immediate code and target code optimizations) are expected to be done in further steps for improving the performance of the system.

4.4 Software System Attributes

The following attributes are what we expect for the future Pangu compiler.

4.4.1 Reliability

Pangu compiler must be able to correctly recognize and convert the Pangu code into target code, despite length and indentation of the Pangu code. No ambiguity in the Pangu code is allowed. The compiler needs to define whether an input code is valid or contains error. Only completely valid code will be converted to target code. Whenever an error is found in the input Pangu code, the compiler must terminate and report its type and position to users.

4.4.2 Availability

Pangu compiler must be available to users all the time. When the Pangu compiler is updated to a new version, its old version still needs to be available to users and able to work correctly.

4.4.3 Security

The whole transformation detail in Pangu compiler must be protected from modifying by users. Only an application programming interface for input Pangu code will be provided. Users must not have any other access to the source code.

4.4.4 Maintainability

All versions of Pangu compiler need to be carefully documented. It needs to have a logging to keep track of bugs encountered by users. It needs to allow recovery from exceptional events without intervention by technical support staff [3].

4.4.5 Portability

Pangu compiler should be able to work on Mac, Linux and Windows platforms. Currently, as the project only reached syntax and semantics checking stage, and python ply is using for the current stage, it can work on all the platforms with python installed.

4.5 Functional Requirements

Currently, we have completed lexical, syntax and semantic modules for Pangu Compiler. Hence, the following use case specification section presents the functional requirements for Pangu Compiler that we have done so far.

4.5.1 Use Case for Writing Source Code

Use Case	Writing Source Code
Actors	Users, asc file (pangu source code file with extension .asc)
Goal Description	Users can write a pangu source code in an asc file, then run the file in the command line.
Precondition	Users have downloaded the Pangu compiler and PLY(Python Lex-Yacc).
Step-by-Step Description	1. Users open a txt file 2. Writing source code in the txt file 3. Saving the source code file with .asc extension 4. Opening the terminal 5. Running program in command line
Postcondition	Users saved the source code in asc file format.
Exceptions	Users run the source code file with the correct format in the command line.

4.5.2 Use Case for Compiling

Use Case	Compiling
Actors	Terminal, and the Pangu compiler
Goal Description	The final goal is that the compiler will translate the source code to the immediate code PIC(a lower level intermediate code). At this stage, it will only display parse trees, production, and symbol tables since we only finished building the lexical, syntax and semantic modules for Pangu Compiler.
Step-by-Step Description	All the Steps are done by the compiler: 1. A parser generated by PLY (Python Lex-Yacc) will translate Pangu source code into an abstract syntax tree and display it in the terminal. 2. It also gives the production of parsing bases on the Pangu Syntax. 3. Displaying the symbol table for the semantic module.
Exceptions	The source code follows the Pangu syntax and semantic module, otherwise errors will arise in the terminal.

5. Software Design/Architecture

[Amy is responsible for Interface viewpoint;
Yujing is responsible for Structure viewpoint;
Wei is responsible for Interaction viewpoint;
Steve is responsible for Algorithm viewpoint;
Baikai is responsible for Resource viewpoint]

5.1 Interface Viewpoint

5.1.1 Software Interface

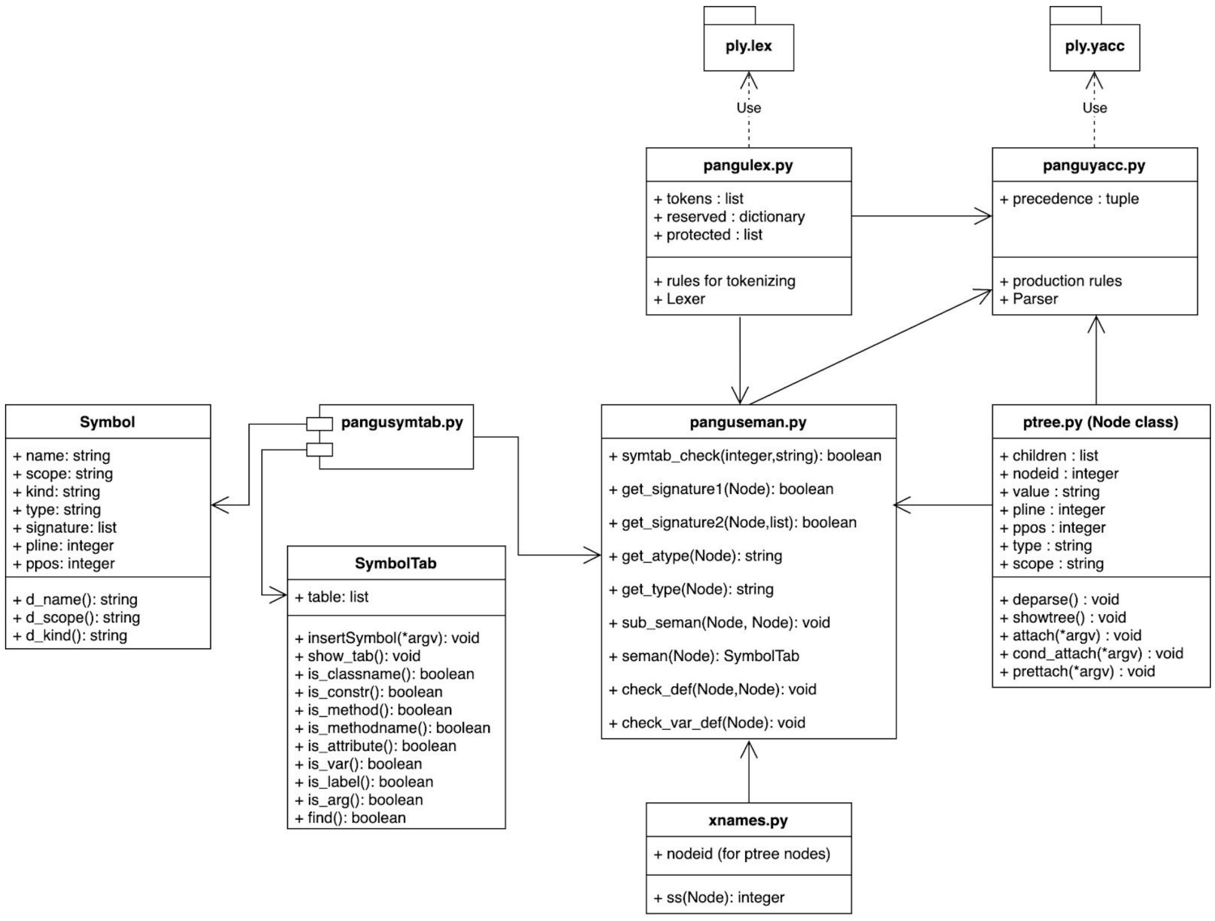
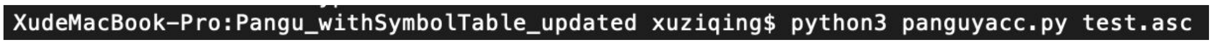


Figure 1.1 : Software Interface

The above figure 1.1 shows a set of interface specifications for each entity required in the program and how the cooperating entities will interact with each other. The three main modules of our program are ***pangulex***, ***panguyacc*** and ***panguseman*** which are for tokenizing, parsing and semantic checking respectively. Also, we used the PLY parsing tool in *pangulex* and *panguyacc* modules. In addition, *pangusymtab*, *xnames* and *ptree* modules are required for implementing the lexer, parser and semantic checking. More details will be discussed in the structure and interaction viewpoint section.

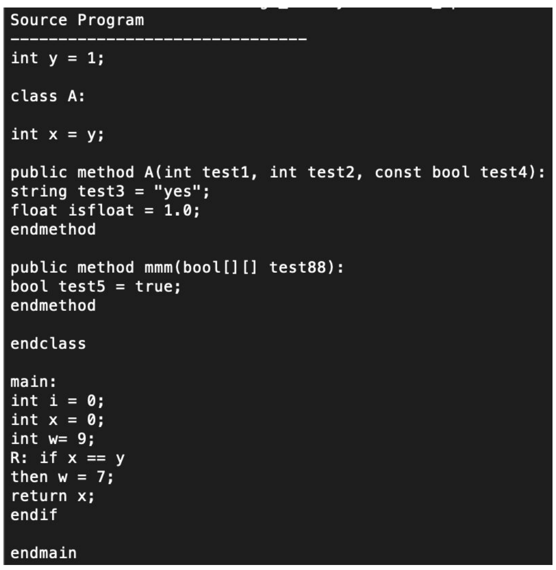
5.1.2 Command Line Interface (CLI)



```
XudeMacBook-Pro:Pangu_withSymbolTable_updated xuziqing$ python3 panguyacc.py test.asc
```

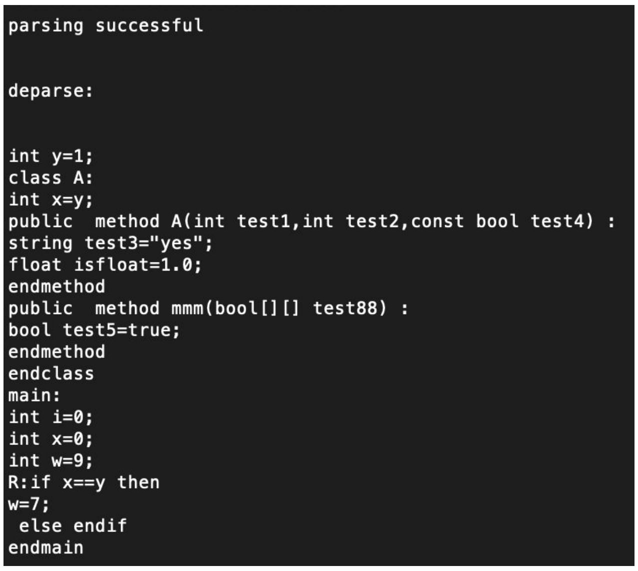
Figure 1.2 : command line interface

The above figure 1.2 shows how to run the program. Open terminal and go to the directory which contains all the program source code. Then run *panguyacc.py* with the command line argument *test1.asc* which is the name of the pangu source file. The content of the output will be different based on the flag which can be modified in the panguyacc.py. For example,



```
Source Program
--------------------------------
int y = 1;

class A:

int x = y;

public method A(int test1, int test2, const bool test4):
string test3 = "yes";
float isfloat = 1.0;
endmethod

public method mmm(bool[][] test88):
bool test5 = true;
endmethod

endclass

main:
int i = 0;
int x = 0;
int w= 9;
R: if x == y
then w = 7;
return x;
endif

endmain
```

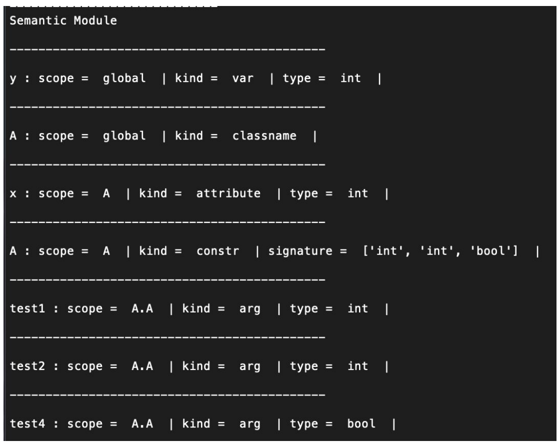
Figure 1.3 : Pangu Source Code



```
parsing successful


deparse:


int y=1;
class A:
int x=y;
public  method A(int test1,int test2,const bool test4) :
string test3="yes";
float isfloat=1.0;
endmethod
public  method mmm(bool[][] test88) :
bool test5=true;
endmethod
endclass
main:
int i=0;
int x=0;
int w=9;
R:if x==y then
w=7;
 else endif
endmain
```

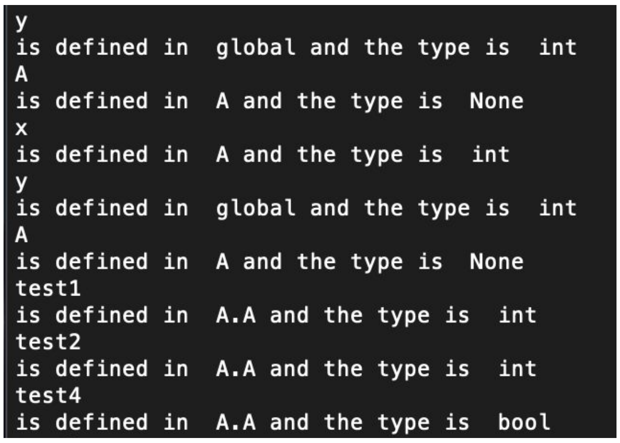
Figure 1.4 : Source Code deparsed from AST

The origin source code (figure 1.2) and deparsed code (figure 1.3) from the AST could be displayed for checking the parsing step if ***y_deparse_ptree*** flag is set to True.



```
Semantic Module

--------------------------------------------

y : scope =  global  | kind =  var  | type =  int  |

--------------------------------------------

A : scope =  global  | kind =  classname  |

--------------------------------------------

x : scope =  A  | kind =  attribute  | type =  int  |

--------------------------------------------

A : scope =  A  | kind =  constr  | signature =  ['int', 'int', 'bool']  |

--------------------------------------------

test1 : scope =  A.A  | kind =  arg  | type =  int  |

--------------------------------------------

test2 : scope =  A.A  | kind =  arg  | type =  int  |

--------------------------------------------

test4 : scope =  A.A  | kind =  arg  | type =  bool  |
```

Figure 1.5 : Symbol Table



```
y
is defined in  global and the type is  int
A
is defined in  A and the type is  None
x
is defined in  A and the type is  int
y
is defined in  global and the type is  int
A
is defined in  A and the type is  None
test1
is defined in  A.A and the type is  int
test2
is defined in  A.A and the type is  int
test4
is defined in  A.A and the type is  bool
```

Figure 1.6

The above figure shows part of the symbol table (figure 1.4) , where the variable is defined in the pangu program and what type the variable is for checking the semantic module (figure 1.5) since ***y_show_symbol_table*** flag is set to True. Also, the AST can be displayed if the variable **y_show_ptree** flag in panguyacc.py is set to True.

5.2 Structure Viewpoint

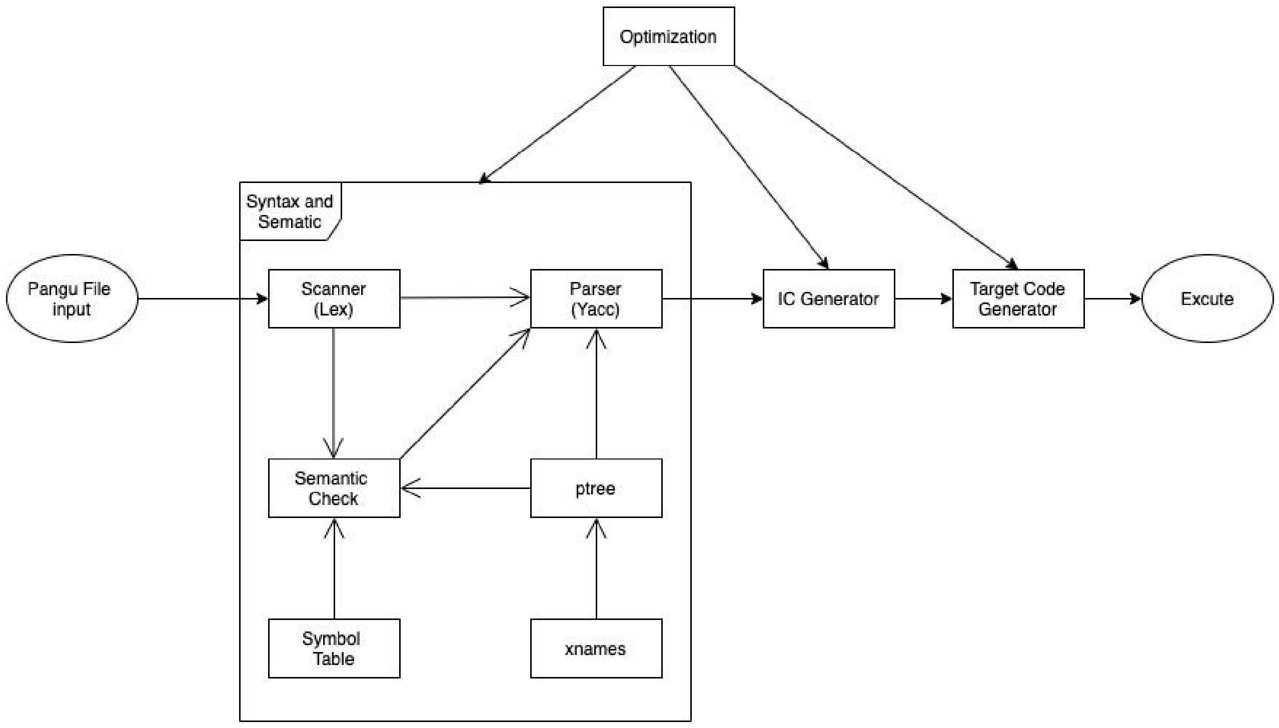


Figure 2: Structure Viewpoint

Internally, a Pangu compiler includes the syntax and semantic check, intermediate code(IC) generator, target code generator and optimization. Our project currently covers the syntax and semantic check part of the Pangu compiler, which is shown in the frame of the above figure. The semantic check module still needs further implementation.

The function of each module is described as follow:

- Scanner(Lex): Lex is used to tokenize an input string. For example, if you have a line of pangu code “int out = 1;”, you need to split it into individual tokens {“int”, “out”, “=”, “1”, “;”}. Python RE is used to recognize the token.
- Parser(Yacc): Yacc is used to analyze the grammar structure of Pangu code, and record the structure for future use.
- ptree: ptree is an abstract syntax tree built to record the grammar structure of the Pangu code.
- xnames: xnames converts a grammar type name to integer.
- Symbol Table: Symbol Table is used to record the variables information of Pangu code. Each variable will have its id, scope, kind, type, signature and position information recorded.

- Semantic Check: Semantic check is used to analyze the semantic aspect of the source code. Including checking and determining scopes of names , binding of names, types of expressions, call signatures, flow-of-control, etc.

5.3 Interaction Viewpoint

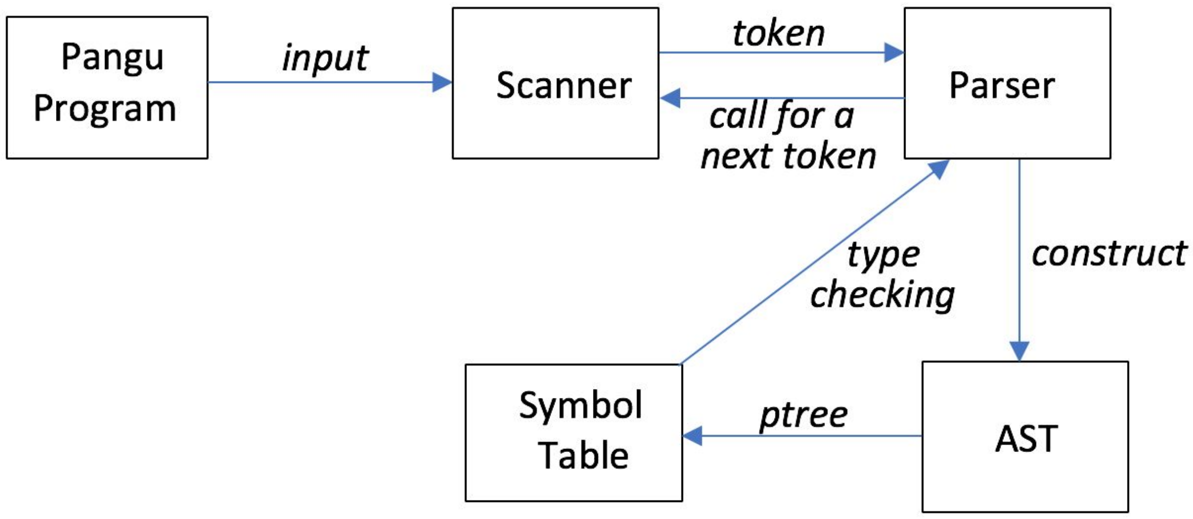


Figure 3: Interaction Viewpoint [1]

Figure 3 shows the interactions among different components of the program. Scanner takes a Pangu program as input and generates tokens for Parser and Parser can call Scanner to get the next token. While parsing, Parser constructs an Abstract Syntax Tree (AST), which can be used to generate a Symbol Table. The Symbol Table is used for type checking by the Parser.

5.4 Algorithm Viewpoint

<u>Panguseman.py</u>
Symtab_check() performs a check on id and will return True if the symbol already exists and is within the current scope. Otherwise, it returns False.
Get_sign() and get_sign2() returns the signature of the node.
Sub_seman() takes care of managing the scope_stack array and also checks the validity of identifiers based on its scope and existence in the symbol table.

[1] Partially from the unpublished book, *Designing and implementing a compiler,* by Frantisek Franek and Michael Liut at McMaster University.

Pangulex.py
Is_protected() and is_reservered() both checks the inputs againsts reserved or protected strings and returns a boolean.
Pangulex.py contains many functions that identify the different types of tokens. For example, t_ASSIG(t) will match with the “=” and will print the token if needed. Printing of tokens is used primarily for code testing.

Panguyacc.py
panguyacc.py‘s goal is to parse the tokens that are coming in from the scanner into an Abstract Syntax Tree.
Each method builds one specific node of the tree and updates the pline and ppos value of each node accordingly. Some methods also update the value of nodes with typing information and children.

Ptree.py
This contains the class Node and its methods. There are a couple of show methods that are used for printing out the parsetree:
attach(self,*argv): attaches all nodes as children
cond_attach(self,*argv): only attaches nodes that are leaves (no children)
prettach(self,*argv): removes all nodes(children)
deparse(self): checks the id of the node and builds a parse tree using tokens. It also calls upon dd(self) and dd1(self,i) to break down expressions

SymbolTab.py
Pangusymtab.py contains classes Symbol and SymbolTab.
Symbol requires the following attributes to be created: name, scope and kind. It can also include type,pline and ppos.
SymbolTab is the class object for the symbol table. The insertSymbol() function is for new entries in the symbol table and showtab() will print the table. SymbolTab also contains a variety of methods that can check different properties of objects and will return the entry’s index if found. It can check if a classname is valid, if a variable is within scope, if a method is valid and more.

xnames.py
It translates grammar types into an integer.

5.5 Resource Viewpoint

There are the resources we used to develop our project:

1. Python
    a. Python is a high level programming language that we use to design the lexical, syntax and semantic modules of Pangu Compiler.
2. System-specific parameters and functions module in Python (sys)
    a. This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.
    b. More specifically, we only used **sys.exit(0)** in our programming. It means exit from Python. This is implemented by raising the SystemExit exception, so cleanup actions specified by finally clauses of try statements are honored, and it is possible to intercept the exit attempt at an outer level.
3. Python Lex-Yacc (PLY)
    a. PLY is an implementation of lex and yacc parsing tools for Python.
    b. The reason why we choose PLY for our compiler is that PLY uses LR-parsing which is reasonably efficient and well suited for larger grammars, and it provides most of the standard lex/yacc features including support for empty productions, precedence rules, error recovery, and support for ambiguous grammars.

6. Code Guide

[all members are responsible for this part]

6.1 Prerequisites

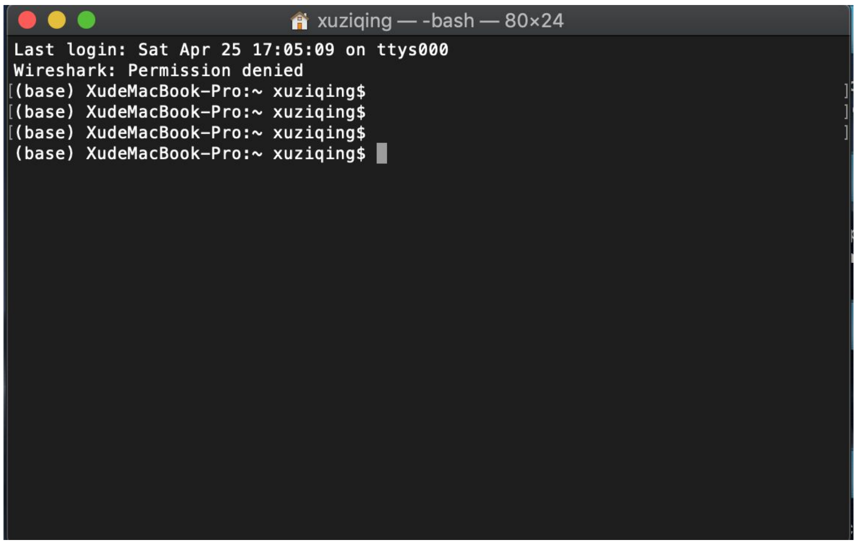
The following are the softwares, tools or packages needed to be downloaded or/and installed in order to run the program.

- PLY - 3.11 (Python Lex-Yacc) parsing tool
    - version 3.11
    - PLY parsing tool is required to implement the lexical and syntax modules of the project. Thus, PLY is required to be downloaded for running the program
    - Here is the link of the information about how to download the tool: https://www.dabeaz.com/ply/
- Command Line Interface
    - Command Line Interface is needed to run the program
    - For example, on Mac OS, the Mac command line is called “Terminal”; on Windows, it is called “Windows Command Prompt”.

- Python 3.8
    - The whole project is implemented in Python language (version 3.8)

6.2 How to Run

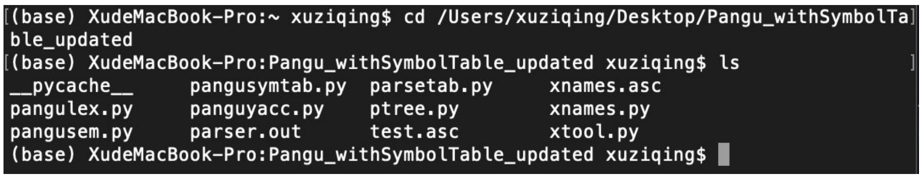
1. Open the CLI on your computer



```
Last login: Sat Apr 25 17:05:09 on ttys000
Wireshark: Permission denied
(base) XudeMacBook-Pro:~ xuziqing$
(base) XudeMacBook-Pro:~ xuziqing$
(base) XudeMacBook-Pro:~ xuziqing$
(base) XudeMacBook-Pro:~ xuziqing$
```

Figure 4.1 : CLI

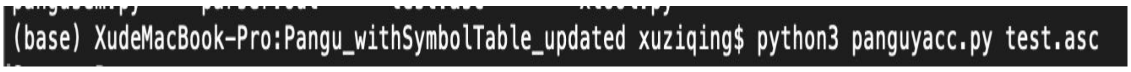
2. Go to the directory which contains all the source code of the program



```
(base) XudeMacBook-Pro:~ xuziqing$ cd /Users/xuziqing/Desktop/Pangu_withSymbolTa
ble_updated
(base) XudeMacBook-Pro:Pangu_withSymbolTable_updated xuziqing$ ls
__pycache__     pangusymtab.py  parsetab.py     xnames.asc
pangulex.py     panguyacc.py    ptree.py        xnames.py
pangusem.py     parser.out      test.asc        xtool.py
(base) XudeMacBook-Pro:Pangu_withSymbolTable_updated xuziqing$
```

Figure 4.2 : Targeted Directory

3. Enter the command line as the following:



```
(base) XudeMacBook-Pro:Pangu_withSymbolTable_updated xuziqing$ python3 panguyacc.py test.asc
```

Figure 4.3 : Command

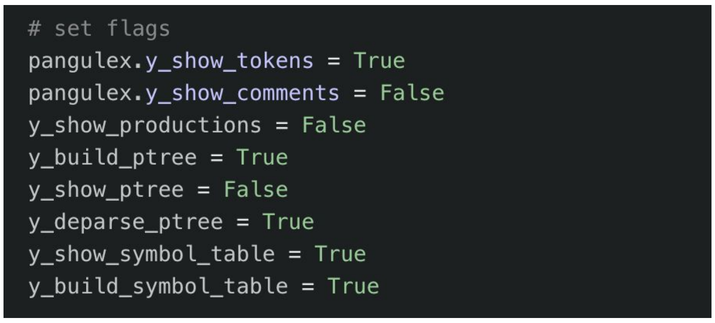
6.3 Syntax of the Command Line

[python3 panguyacc.py pangu_s_c]

where pangu_s_c is the name of the pangu source code file

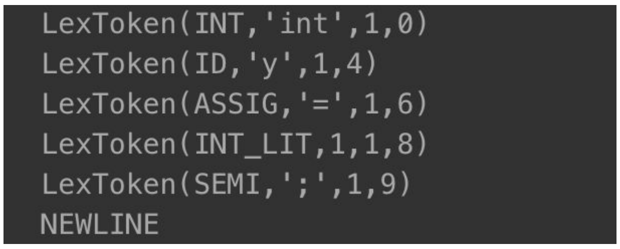
6.4 Testing Flags

Several testing flags are available, they can be switching on in panguyacc.py



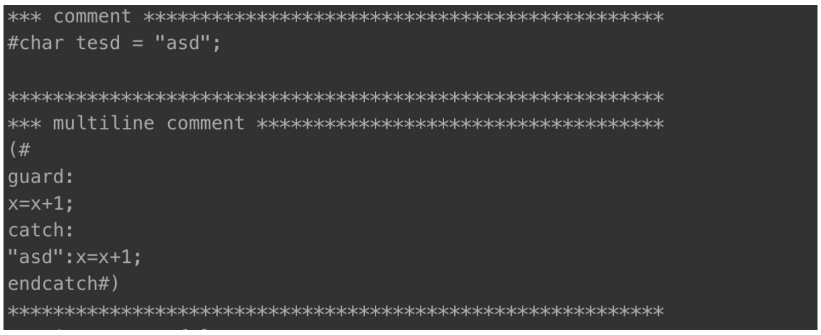
```
# set flags
pangulex.y_show_tokens = True
pangulex.y_show_comments = False
y_show_productions = False
y_build_ptree = True
y_show_ptree = False
y_deparse_ptree = True
y_show_symbol_table = True
y_build_symbol_table = True
```

1. Show_tokens: lexical token in the pangu source code will be printed.



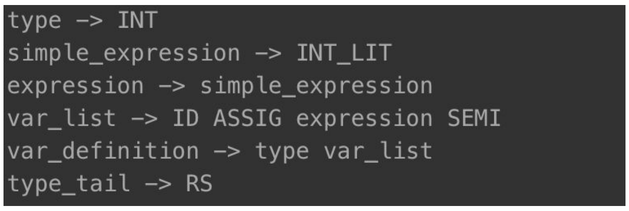
```
LexToken(INT,'int',1,0)
LexToken(ID,'y',1,4)
LexToken(ASSIG,'=',1,6)
LexToken(INT_LIT,1,1,8)
LexToken(SEMI,';',1,9)
NEWLINE
```

2. Show_comments: comments will not be hidden in the depares code.



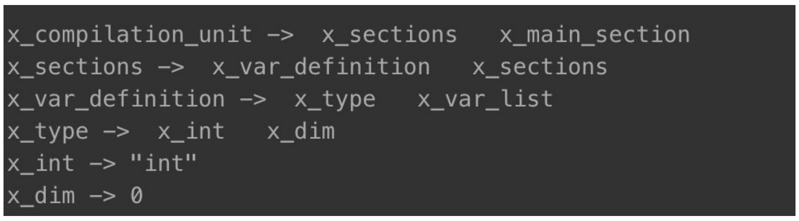
```
*** comment ************************************************
#char tesd = "asd";

************************************************************
*** multiline comment **************************************
(#
guard:
x=x+1;
catch:
"asd":x=x+1;
endcatch#)
************************************************************
```

3. Show_productions: grammar production will be printed.



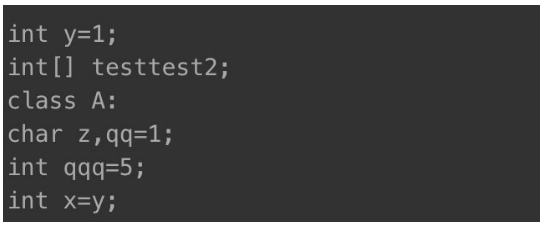
```
type -> INT
simple_expression -> INT_LIT
expression -> simple_expression
var_list -> ID ASSIG expression SEMI
var_definition -> type var_list
type_tail -> RS
```

4. Show_ptree: AST structure will be printed.



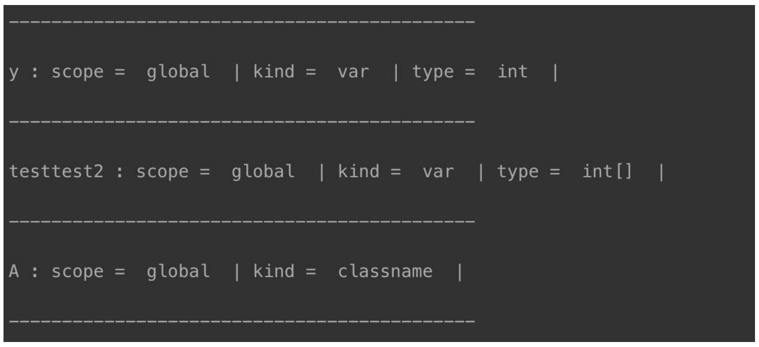
```
x_compilation_unit ->  x_sections   x_main_section
x_sections ->  x_var_definition   x_sections
x_var_definition ->  x_type   x_var_list
x_type ->  x_int   x_dim
x_int -> "int"
x_dim -> 0
```

5. Show_deparse_tree: code built from deparsing AST will be printed.



```
int y=1;
int[] testtest2;
class A:
char z,qq=1;
int qqq=5;
int x=y;
```

6. Show_symbol_table: symbol table will be printed.



```
-------------------------------------------

y : scope =  global  | kind =  var  | type =  int  |

-------------------------------------------

testtest2 : scope =  global  | kind =  var  | type =  int[]  |

-------------------------------------------

A : scope =  global  | kind =  classname  |

-------------------------------------------
```