X0 Specification

Yingzhe Lyu, 10152130255

November 28, 2018

1 Introduction

X0 is a C-like programming language used in this course project of Compiler Principle Practice which capable of calling functions with arguments and a return value, declaring multi-dimension arrays, and other functionalities that a typical procedure-oriented language supposed to have (except float-point numbers). Based on this language, we have also designed a complete compiler (I would rather like refer it to an interpreter) with modern GUI.

2 X0 Language Specification

2.1 Usage

First of all, please note that our implementation completely separate interpreting from compiling and the interpretation part is integrated in the GUI. Therefore, to build up the executable compiler program, you need to build up both the compiler (written in C with Flex and Bison) and the GUI (written in Python with PyQt5). If you have Flex, Bison, Make for Windows and Python (with PyQt5 package) on you Windows OS computer, and suppose the source file directory is `<source path>`, you can manually build it as follow:

```
> cd <source path>
> make
> python main.py
```

Or, if you failed to do so, you may run the pre-compiled `X0Rush.exe` GUI executable for X86-64 Windows OS. Please note that the name of compiler binary file must be `x0.exe` and be in the same directory as GUI binary, otherwise you can only edit but not able to compile/debug the source files.

2.2 Implemented Extra Functionalities

The features beyond the basic requirement but in optional requirement, and their correspondent bonus are listed here:

Extra Point	Bonus	Comment
Time Bonus	6	Second Week
GUI and Debug	10	
Module Division Operator %	2	
Logical Operator XOR	2	
Even-Odd Operator ODD	2	
Increment and Decrement Operators	2	
Switch Statement	5	
For Statement	0	Obligatory Point
Break, Continue and Exit Statement	6	
Do-While Statement	2	
Repeat-Until Statement	2	
Boolean Data Type	10	
Constant Variable	4	
Multi-Dimension Array	10	
Function Call	8	
Arguments and Return Value for Functions	15	
Total	86	

2.3 Keywords

As X0 is a case sensitive language, all keywords must be written in lowercase. Here is a list of all keywords preserved in our implementation. (You may suppose `odd` should appear here as a keyword, but actually, we treat it as an unary operator and therefore listed in the operators and precedence section)

break	bool	case	char	const	continue
default	do	else	exit	false	for
if	int	main	read	repeat	return
switch	true	until	void	while	write

2.4 Identifiers

All identifiers in a X0 program, including both variables and function names, should only contain letters (both uppercase and lowercase), digits and underscores and the first letter of an identifier must be either a letter or an underscore. Please also note that identifiers must be unique, as they are created to give unique name to an entity to identify it during the execution of the program. Also remember, identifier names must be different from keywords, as you are not allowed to use preserved keyword as an identifier.

2.5 Operators and Precedence

The list of all operators and their precedence is given here.

Precedence	Operator	Description	Associativity
1	[]	Array subscripting	Left-to-right
2	++ −−	Suffix increment and decrement	
	()	Function call	
3	++ −−	Prefix increment and decrement	Right-to-left
	+ −	Unary plus and minus	
	!	Logical NOT	
	odd	Odd-even judgment	
	(*type*)	Type conversion	
4	∗ / %	Multiplication, division, and remainder	Left-to-right
5	+ −	Addition and subtraction	
6	< <=	For relational operators $\prec$ and $\preceq$ respectively	
	> >=	For relational operators $\succ$ and $\succeq$ respectively	
7	! = ==	For relational = and $\neq$ respectively	
8	&&	Logical AND	
9	||	Logical OR	
10	=	Simple assignment	Right-to-left

2.6 Type Casting

A type cast is basically a conversion from one type to another. Two types of type cast are in our X0 implementation:

1. **Implicit Type Conversion**
   Also known as automatic type conversion, which is done by the compiler on its own without any external trigger from the user. Generally, it takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lost of data. All the data types of the variables are upgraded to the data type of the variable with largest data type as follow: `boolean->character->integer`.

2. **Explicit Type Conversion**
   This process is also called manual type casting and it is user defined. Here the user can type cast the result to make it of a particular data type. The syntax in X0 is: `(type) expression`. See operator and precedence section for its precedence information.

2.7 Special Rules

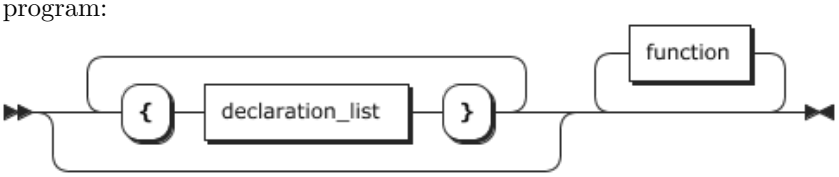
- Special write statements: WRITE STRING ';' for string output and WRITE ';' for CRLF output.
- Return value from read statements: our read statement will return the number just read in.
- Variable scope, allows same identifier as local variables in different functions and defining of global variables.
- Immediate numbers, which allows bare number/character/boolean value appeared in expressions.

3 Parser Specification

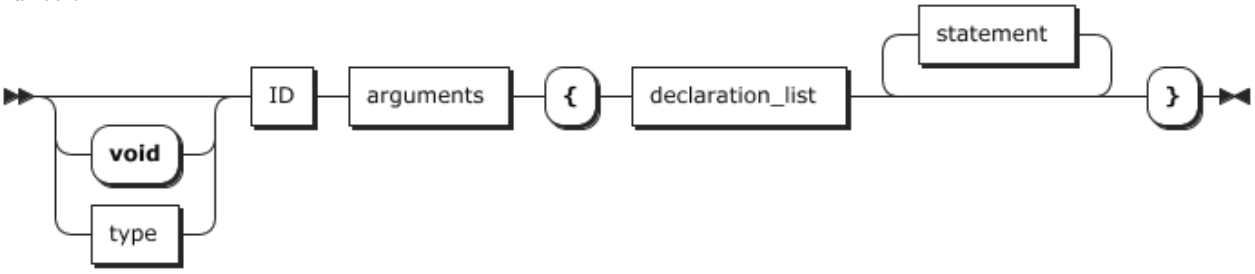
3.1 Railroad Diagram

In order to reduce the length of this **lengthy** report, we replace the productions in .y file with simplified ones, while recognizing the same language. Please note that only grammar rules are displayed in this diagram, semantic restrictions cannot be listed here and can be found in the following sections. Although Flex & Bison have the capacity to handle operator precedence implicitly, we choose to explicitly do this in grammar layer for the convenience of semantic actions.

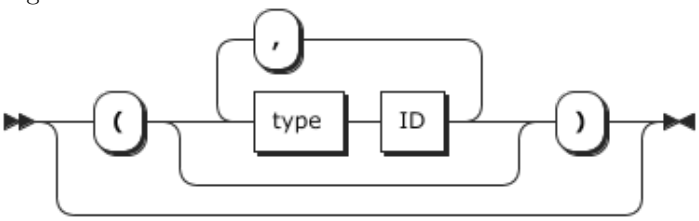
1. program:



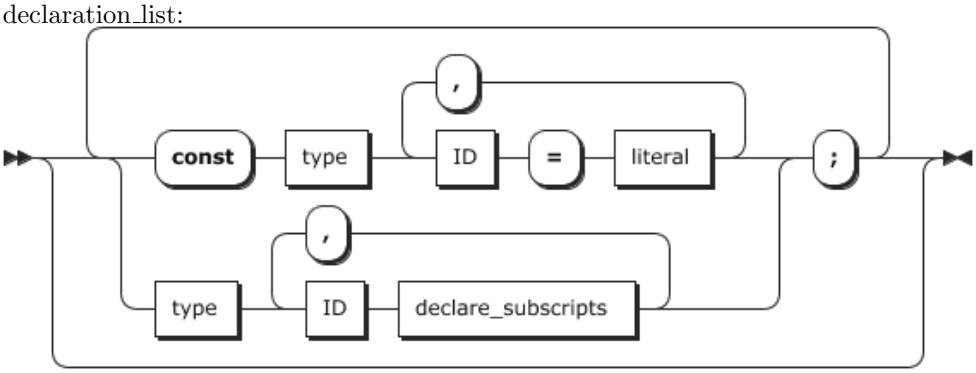
2. function:



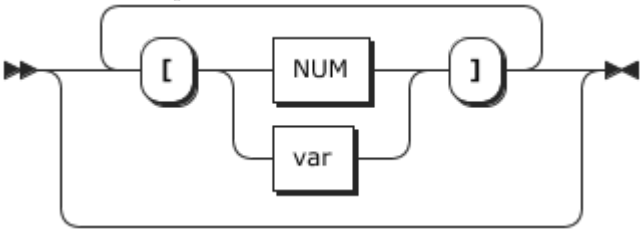
3. arguments:



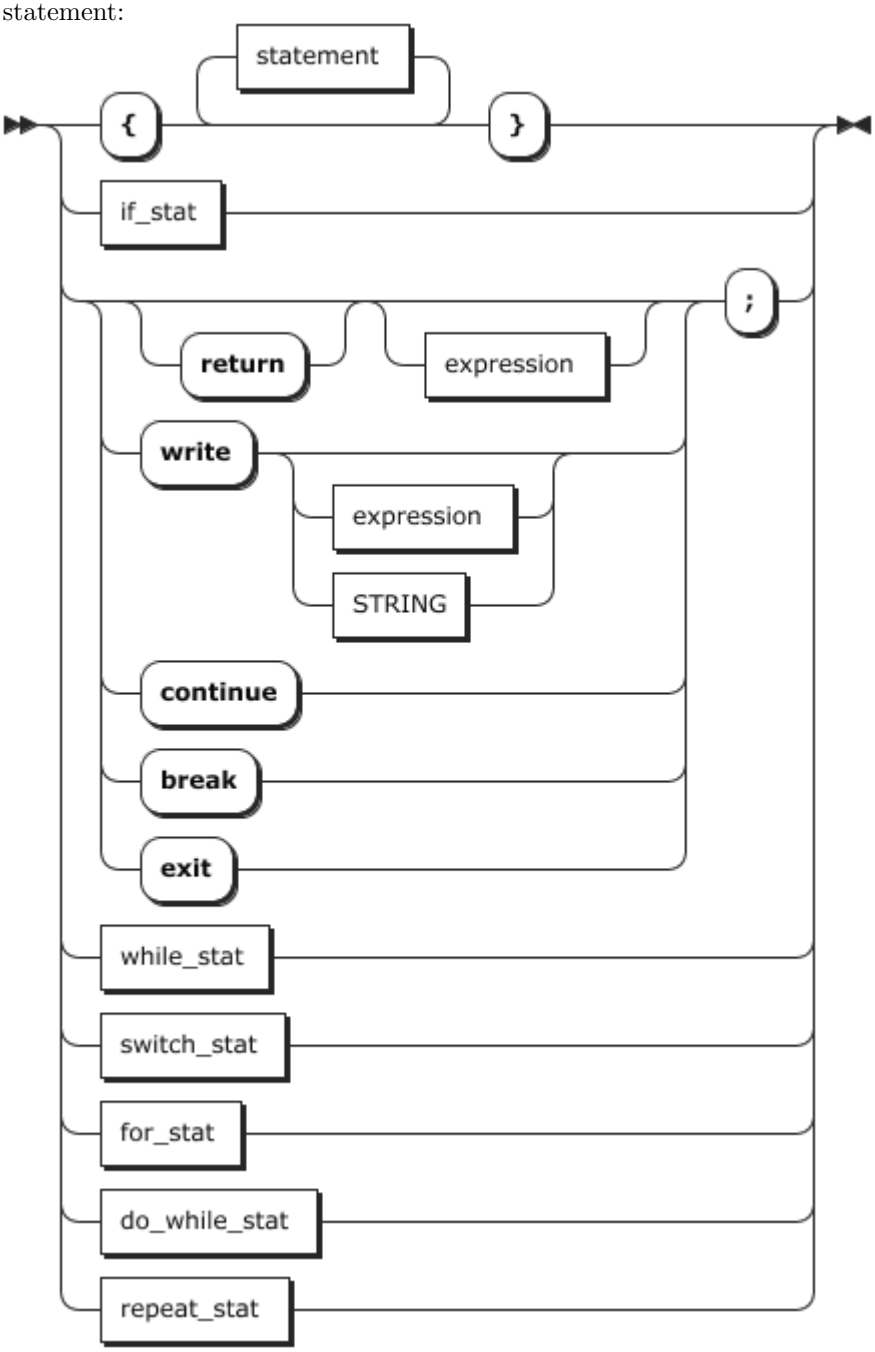
4. declaration_list:



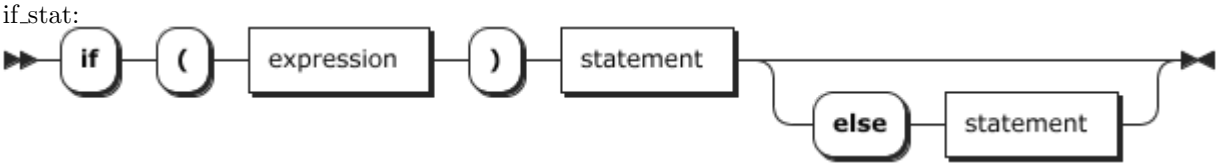
5. declare_subscripts:



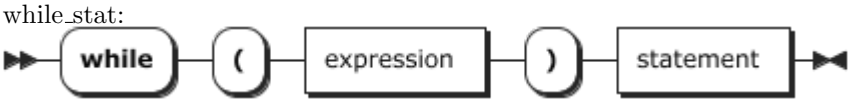
6. statement:



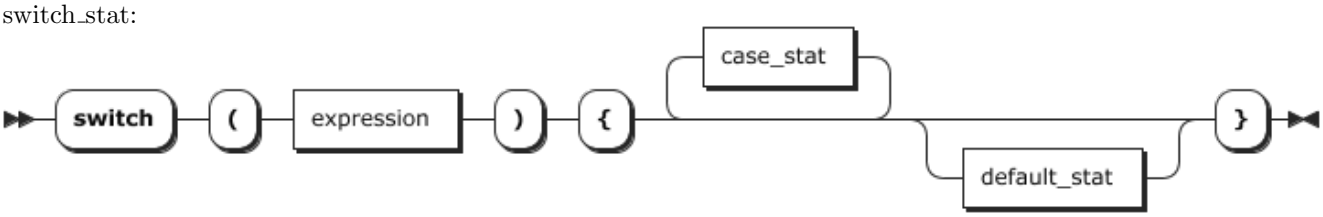
7. if_stat:



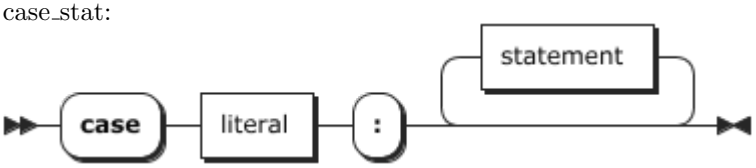
8. while_stat:



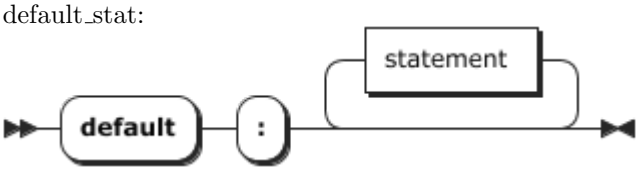
9. switch_stat:



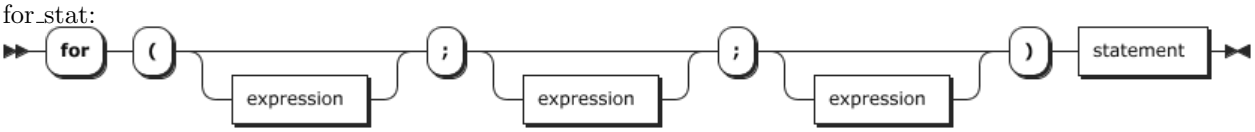
10. case_stat:



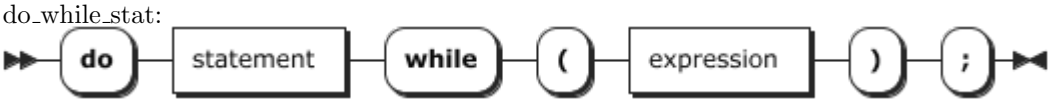
11. default_stat:



12. for_stat:

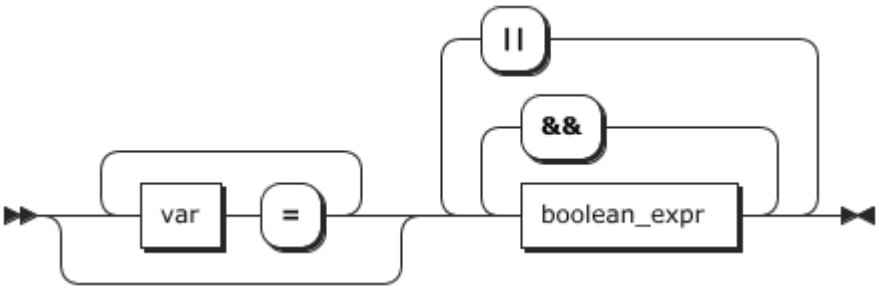


13. do_while_stat:

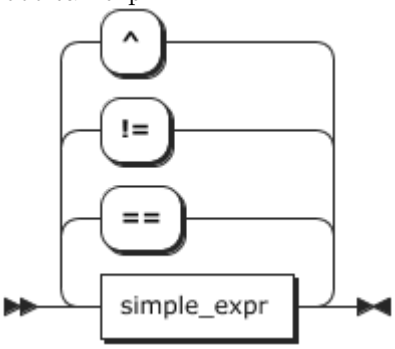


14. repeat_stat:

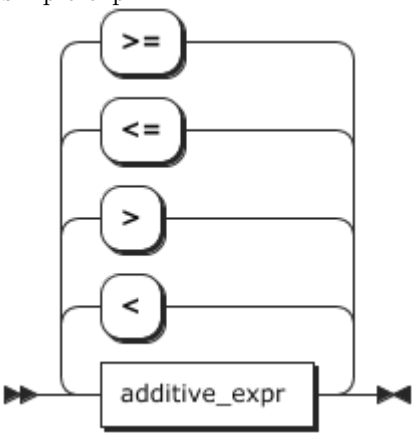
15. expression:



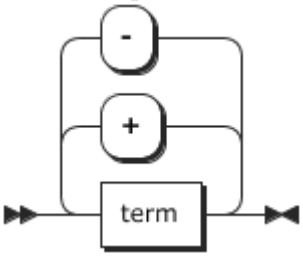
16. boolean_expr:



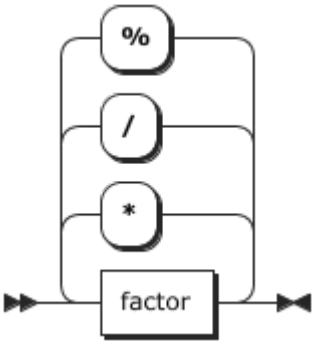
17. simple_expr:



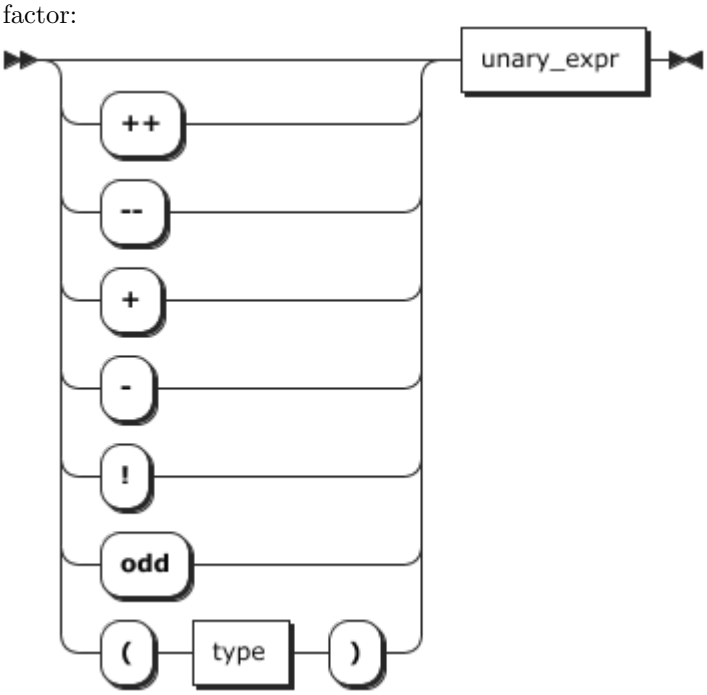
18. additive_expr:



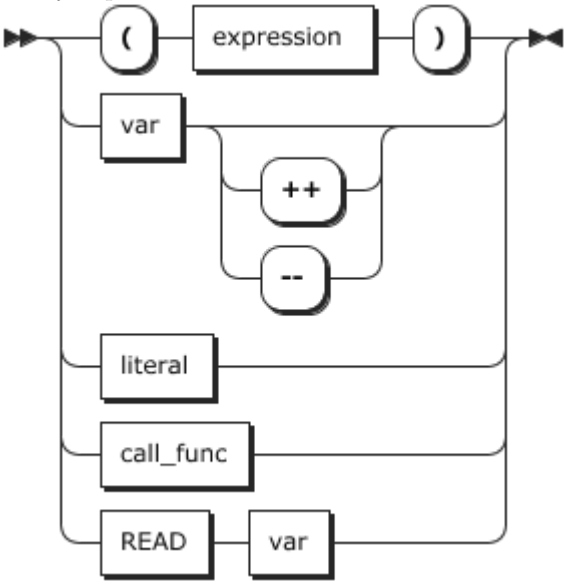
19. term:



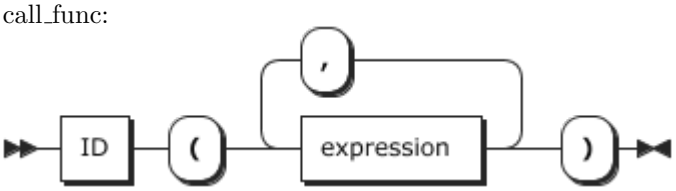
20. factor:



21. unary_expr:

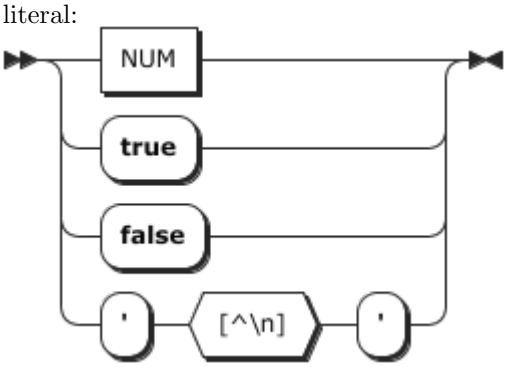


22. call_func:

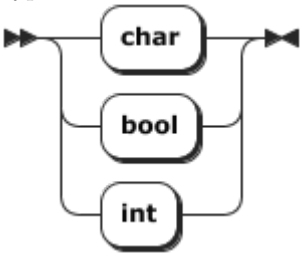


23. var:

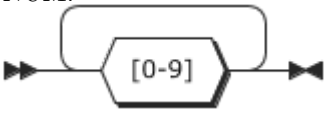
24. literal:



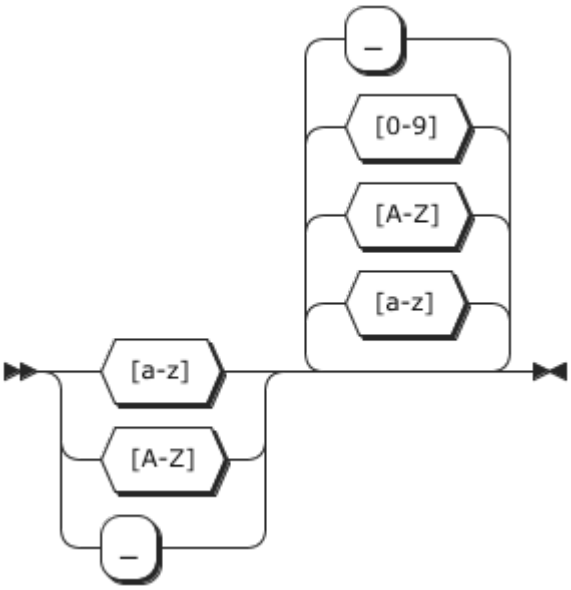
25. type:



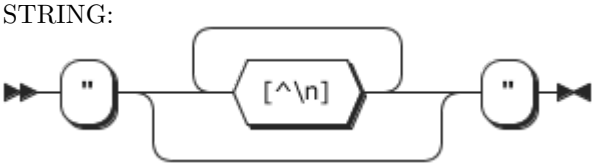
26. NUM:



27. ID:



28. STRING:



3.2 LL(1) Validation

According to the two validating rules of LL(1) grammar judgment, we can assert that our grammar is LL(1) from the grammar diagram above (however, we choose to complete the compiler with Flex & Bison, thus the grammar is automatically implemented in LALR(1), this validation have mere usage). In addition, this grammar is ambiguous (e.g. the dangling-else problem) if without the restrictions listed below, please refer to the following section for our solution.

3.3 Semantic Restrictions

1. **Break/Continue Statements**
   It's clear that break/continue statements should only appeared inside a loop-statement, except that break statements can also appeared in switch-statements. Considering it's trivial to write this restriction into grammar rules, we check the context in Bison once found a break/continue statement to guarantee that it appears in the right position.

2. **Reference of Variables**
   If you try to use a variable that is not declared yet, or not in current scope, the compiler should dump an error message on it. Therefore, we check the symbol table every time a variable is referred in order to ensure that it is available.

3. **Array Dimension Limitation**
   Although theoretically, a user can declare an array with as many dimensions as he/she want, but the limitation of our implementation prevent users from doing so. The maximal dimensions of a single array is 100, trying to create an array with more than 100 dimensions will trigger an error. Such restriction won't cause big problem since nobody would like to manage an array with so many anonymous dimensions.

4. **Type Capability**
   We do a lot of type checks when declaring a variable, doing arithmetic operations, passing a parameter, returning a value, etc. Please see the error table for more detail.

5. **Main Function Existence**
   You may note that we don't required a main function in grammar rule, since we handle main function in the same way as other functions. But actually, a main function is a must and the compiler will check the existence of it then do some relevant actions at the end of compiling. If there's no main function declared in an X0 program, an error message will be generated.

6. **Solution for Dangling-Else**
   Dangling-else is a classic problem in compiler design, which refer to the situation that an else-clause can't distinguish which if-clause to pair with, for example: `if (cond1) if (cond2) stat1; else stat2;`. In this example, the else-clause can either pair with the first if-clause or the second one. We choose to use the same solution of C language: always pair an else-clause with the nearest if-clause above. Therefore, we manually specified that the priority of shifting an else-clause is higher than reducing (which means our compiler prefer to form a if-else-clause rather than a standalone if-clause).

3.4 Error Table

This table listed all kinds of error message that may be generated by our compiler. Please note that type 0 refers to grammar(syntax) errors and other types refer to correspondent semantic in semantic restrictions section.

Type	Error Message
0	In line *line_no*: error: syntax on *token*
1	Continue statement outside of loops
1	Break statement outside of loops or switch
2	Duplicate identifier
2	Not a modifiable lvalue
2	Symbol should be a function
2	Write data to a constant value
2	Function has not been declared
2	Variable has not been declared
2	Function cannot act as a rvalue
2	Subscription dimension not correct
2	Subscript operation on a non-array variable
2	Fetch array size from a non-positive literal number
2	Argument type not compatible with function declaration
2	Fetch array size from a non-constant or non-positive variable
2	Argument number not correspond with the declaration of called function
3	Unable to handle an array with over 100 dimensions
4	Return nothing in a non-void function
4	Return a value in a void function
4	Void type cannot act as a lvalue
4	Void type cannot act as a operand
4	Function type cannot act as a operand
5	Main function not found

Specially, we generate warning messages for downward type conversions, but won't forbid such actions. You will see this message in such situation: `In line <line_no>: Warning: downward type conversion may lost precision`

4 Interpreter Specification

4.1 Interpreter Structure

As mentioned previously, our interpreter is separated completely from the parser, only receiving a symbol table file `sym_table` and an intermediate code file `inter_code` from the output of our parser. After loaded the two files (and checked the return value of called parser process to ensure the compilation is succeed), our interpreter, which encapsulated in the GUI, will start build the runtime environment and run the code.

4.2 Runtime Environment

The main component for the runtime environment of our interpreter is a data stack with default size of 40K words, and all the elements are initialized with -1 at the very beginning. Two pointers are attached to the stack to control its status: `base` indicates the base address of current procedure record, and `top` point to the stack top. In addition, `pointer` indicates the executing instruction, and the `status` boolean variable shows whether the program terminates.

4.3 Intermediate Code Format

The intermediate codes in the output file of parser are in following format: `<index> <op> <num1> <num2>`, where `index` indicated the line number starts from 0, `op` is in 8 possibilities described below, and `num1` and `num2` have varied meaning depend on the type of `op`. Here's the all types of `op` and their explanation.

OP	Format	Description
lit	lit 0 a	Push number a on the top of stack
opr	opr 0 0	Return from a void function
opr	opr 1 0	Return from a function with return value
opr	opr 0 1	Negate the stack top
opr	opr 0 2	Perform add operation on two top entry in the stack
opr	opr 0 3	Perform minus operation on two top entry in the stack
opr	opr 0 4	Perform multiplication operation on two top entry in the stack
opr	opr 0 5	Perform division operation on two top entry in the stack
opr	opr 0 6	Perform even-odd judgment (module 2) on two top entry in the stack
opr	opr 0 7	Perform modulation operation on two top entry in the stack
opr	opr 0 8	Perform equality judgment on two top entry in the stack
opr	opr 0 9	Perform inequality judgment on two top entry in the stack
opr	opr 0 10	Perform less-than judgment on two top entry in the stack
opr	opr 0 11	Perform greater-equal judgment on two top entry in the stack
opr	opr 0 12	Perform less-equal judgment on two top entry in the stack
opr	opr 0 13	Perform greater-than judgment on two top entry in the stack
opr	opr l 14	Output and pop stack top according to the value of l (1: as bool, 2: as char, 3: as int)
opr	opr 0 15	Output a newline character (CRLF)
opr	opr l 16	Input a value to stack top according to the value of l (1: as bool, 2: as char, 3: as int)
opr	opr 0 17	Perform NOT operation on two top entry in the stack
opr	opr 0 18	Perform AND operation on two top entry in the stack
opr	opr 0 19	Perform OR operation on two top entry in the stack
opr	opr 0 20	Perform XOR operation on two top entry in the stack
opr	opr 0 21	Perform equality judgment on two top entry in the stack **without pop the first operand**
opr	opr 0 22	Pop the stack top
opr	opr 0 23	Copy current stack top and put it on the top of stack
opr	opr l 24	Pass parameter according to the value of l (1: as bool, 2: as char, 3: as int)
opr	opr l 25	Do type conversion according to the value of l (1: to bool, 2: to char, 3: to int)
lod	lod l a	($l \geq 0$) Load a variable to stack top from layer diff l, address a
lod	lod l a	($l < 0$) Load an array element to stack top from layer diff $-l-1$, address $a +$ `stacktop`
sto	sto l a	($l \geq 0$) Store the value on stack top to variable at layer diff l, address a
sto	sto l a	($l < 0$) Store the value on stack top to variable at layer diff $-l-1$, address $a +$ `stacktop`
cal	cal -1 a	Call main function whose code address is a
cal	cal 0 a	Call a other than function whose code address is a
ini	ini 0 a	Allocate a spaces for local variables
jmp	jmp 0 a	Jump to code address a without condition
jpc	jpc 0 a	Jump to code address a if stack top equals to 0, then pop it

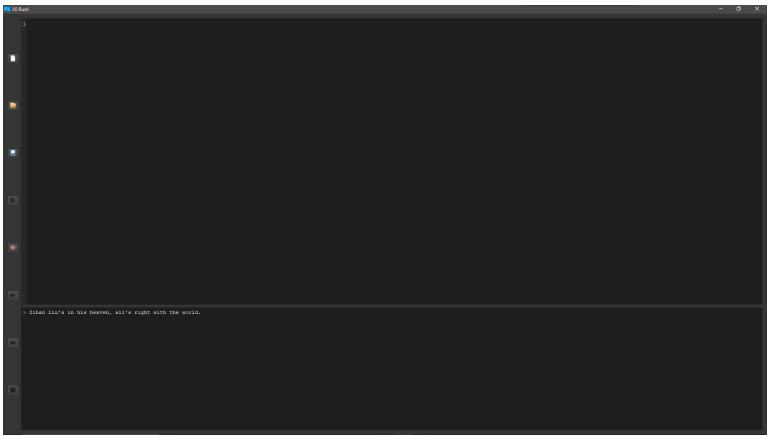
By default, when operating a binary operation, the second operand is the stack top, and the first operand is the element under the stack top. If without special comment, an operation will pop all the operand(s) and push the result on the stack top and store/load won't pop the manipulated value.

4.4 Interpreting Process

Once the GUI executed the compiler `x0.exe` and successfully compiled the source file, symbol table file and intermediate code file will be loaded into the interpreter, and initialization starts. After that, the interpret would step/forward to breakpoint in debug mode, or step until program terminates/encounters an runtime error under the control of GUI.

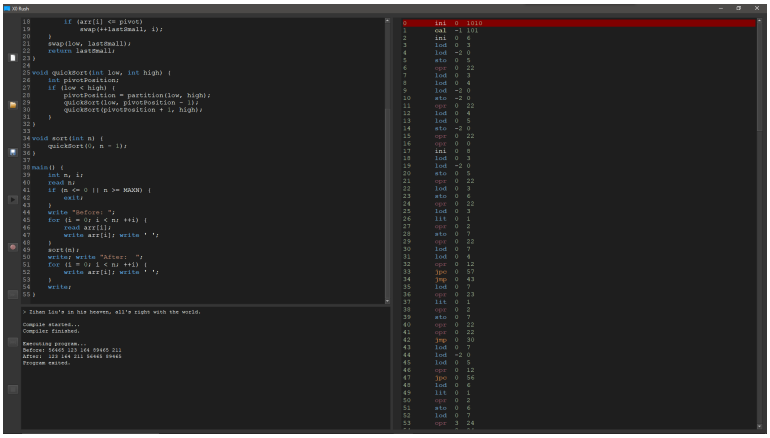
5 GUI Specification

5.1 Initial Window



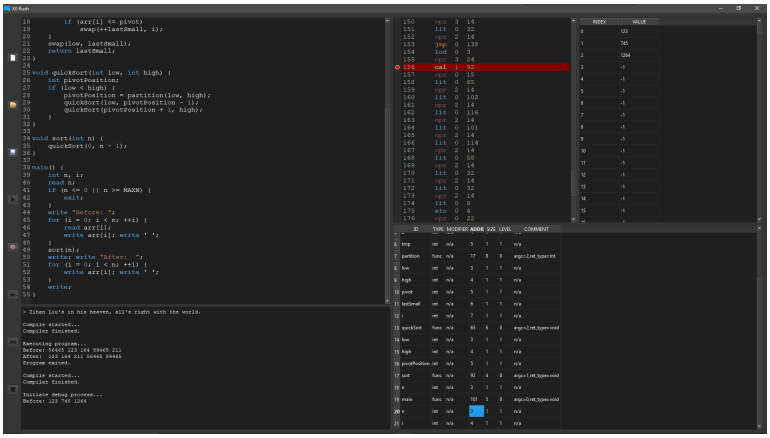
Once you opened our GUI, you'll see this window. At the left edge of GUI, there are several buttons, used to create new file, open file, save file, compile and execute, compile and debug, step debugger, move to next breakpoint, and exit debug respectively. The last three buttons are specially designed for debug process and others are general buttons. In the main zone, above is edit area equipped with code highlight and proper font, feel free to edit your code here, and at the bottom is message area, you would see information and program output here.

5.2 Running Window



On the left is the window after executing a file, you can see the right half of UI now displays the intermediate code and the message area displays message from compiler and program output.

5.3 Debug Window



Left is now the debug window when debugging the same file. On the right, there are symbol table and runtime stack displayed, and you can add breakpoints, use three special buttons in this mode. The red line indicates the next instruction to be executed.