X0_Define_Expension Compiler

软件设计说明书

作者:钱庭涵

版本:v1.0

日期:2018/12/16

1. 介绍

1.1 目的

本项目对给定的 X0 语言进行了词法和语法定义上的扩展，对扩展后的 X0 语言（即 X0_Define_Expension），编写其编译器，实现词法分析、语法分析、出错处理、代码生成和解释程序的功能。

1.2 语言及运行环境

编程语言：Python 3.5

IDE：JetBrains PyCharm

运行环境：Windows 10 64bit

1.3 X0 扩展语言功能

1.3.1 X0 语言文法说明

X0 语言是一种类 C 语言，该语言支持多种变量类型，所有类型在计算时当作整数处理，但输入输出时以该类型形式进行，如字符型的输入输出均为字符型。

标识符格式为：字母打头，后接字母或数字。

注释用“/*”和“*/”括起来，但不能嵌套。词法分析程序并不输出注释，在词法分析阶段，注释的内容将被删掉。

为了从源程序字符流中正确识别出各类单词符号，相邻的标识符、整数或保留字之间至少要用一个空格分开。

1.3.2 扩展前后 X0 语言功能对比

支持功能	扩展前	扩展后增加的功能
变量类型	整型，整型一维数组，字符型，字符型一维数组	布尔型，布尔型一维数组
整型输入值	无符号整数	支持负整数
运算符	<，<=，>，>=，==，!=，+，-，*，/	求余%，异或 XOR，判断整数奇偶 ODD，自增++，自减--
语句	if，if else，while，read，write	switch case，for，continue，exit，break，do while，repeat until
逻辑运算		and，or，not 布尔数据运算
常量		增加常量、过程的定义和使用，区分常量与变量
函数		允许调用（call）过程

2. 编译器系统结构

本编译器使用 python 3.5 编程语言，使用 python 的 PLY 库中的 Lex 实现词法分析，使用 PLY 库中的 Yacc 实现语法分析，使用 wx 库搭建前端页面。

使用 python 实现编译器的文献资料很少，PLY (Python Lex-Yacc)的使用方法可以参考官方文档 http://www.dabeaz.com/ply/ply.html 。

2.1 编译器

2.1.1 Lex 的 token

保留字	main, int, char, if, else, do, while, repeat, until, write, read, XOR, ODD, for, exit, continue, break, switch, case, default, bool, and, or, not, const, procedure, call

其他 token	NUMBER	\d+	EQUAL	=	PLUS	+	MINUS	-
	TIMES	*	DIVIDE	/	LPAREN	(	RPAREN	)
	LBRACE	{	RBRACE	}	LSS	<	LEQ	<=
	GTR	>	GEQ	>=	EQL	==	NEQ	!=
	SEMICOLON	;	LBRACKET	[	RBRACKET	]	COLON	:
	SFPLUS	++	SFMINUS	--	MOD	%		
	ID	[a-zA-Z][a-zA-Z0-9]*						

2.1.2 Yacc 的优先级

```
precedence = (
    ('left', 'and', 'or', 'XOR'),
    ('left', 'LSS', 'LEQ', 'GTR', 'GEQ', 'EQL', 'NEQ'),
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE', 'MOD'),
    ('right', 'SFPLUS', 'SFMINUS', 'not')
)
```

各运算符的优先级从低到高排序，同一个元组内运算符优先级相同，每个元组内第一个字段表示运算符结合方式（左结合、右结合、无结合方式）。

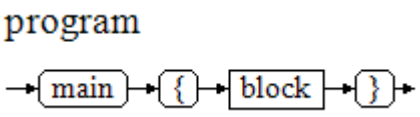
2.1.3 X0 语言语法规则、语法图、语法实现

（语法规则一览见附录 1）

规则 1：

```
program = "main" "{" block "}".
```

语法图：



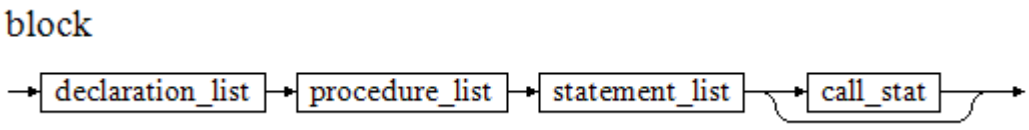
语法实现：

```
program : main LBRACE block RBRACE
```

规则 2：

```
block = declaration_list procedure_list statement_list [call_stat].
```

语法图：



语法实现：

```
block : gen_jmp declaration_list rec_local_adr procedure_list gen_ini
statement_list call_stat
      | gen_jmp declaration_list rec_local_adr procedure_list gen_ini
statement_list
```

说明：

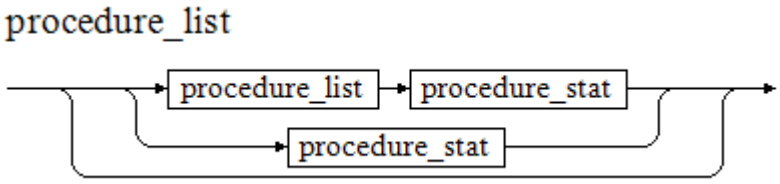
PLY 的 yacc 只能实现从语法分析到得到运行结果，并不能生成目标虚拟机代码指令，因此需要编写其他函数来生成指令。而在进入当前 yacc 程序的 p_block 函数时，已经是移进规约到上述规则的末尾处。想要得到正确的目标代码，必须在上述规则中间，嵌入别的动作来生成相应的指令。gen_jmp 用于生成一条待回填的 jmp 指令，此处的作用为过程跳转；rec_local_adr 用于记录当前层 block 的局部变量地址；gen_ini 用于生成一条 ini 初始化指令。

【注意】要先声明变量和常量，再声明过程（如果有的话），call 函数调用过程需要在一个 block 的最后（如果有的话）。

规则 3：

```
procedure_list = [procedure_list procedure_stat | procedure_stat].
```

语法图：



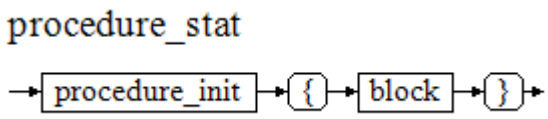
语法实现：

```
procedure_list : procedure_list procedure_stat
               | procedure_stat
               |
```

规则 4：

```
procedure_stat = procedure_init "{" block "}".
```

语法图：



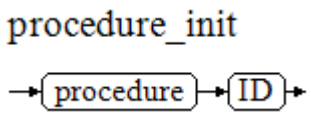
语法实现：

```
procedure_stat : procedure_init LBRACE block RBRACE
```

规则 5：

```
procedure_init = "procedure" ID.
```

语法图：



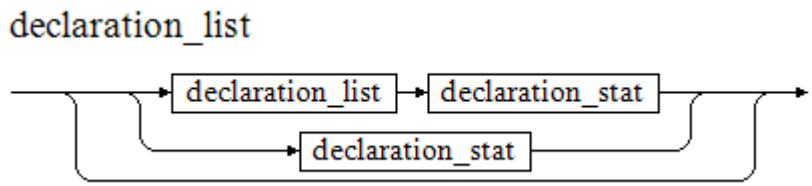
语法实现：

```
procedure_init : procedure ID
```

规则 6：

```
declaration_list = [declaration_list declaration_stat | declaration_stat].
```

语法图：



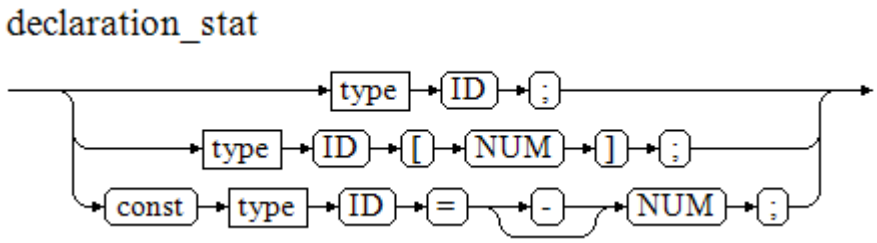
语法实现：

```
declaration_list : declaration_list declaration_stat
                 | declaration_stat
                 |
```

规则 7：

```
declaration_stat = type ID ";" | type ID "[" NUM "]" ";" | "const" type ID "=" ["-"] NUM ";".
```

语法图：



语法实现：

```
declaration_stat : type ID SEMICOLON
                 | type ID LBRACKET NUMBER RBRACKET SEMICOLON
                 | const type ID EQUAL NUMBER SEMICOLON
```

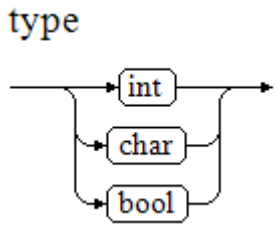
说明：

【注意】定义 constant 常量时需要同时给这个常量赋值，此处只能定义单个常量，不能定义数组常量。

规则 8：

```
type = "int" | "char" | "bool".
```

语法图：



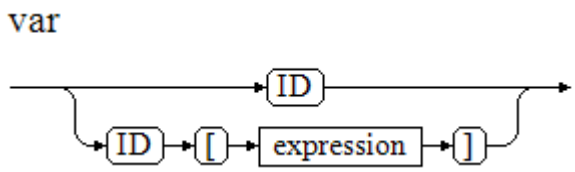
语法实现：

```
type : int
     | char
     | bool
```

规则 9：

```
var = ID | ID "[" expression "]".
```

语法图：



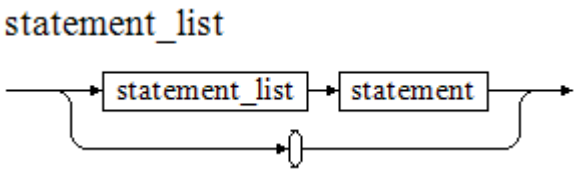
语法实现：

```
var : ID
    | ID LBRACKET expression RBRACKET
```

规则 10：

```
statement_list = statement_list statement | "".
```

语法图：



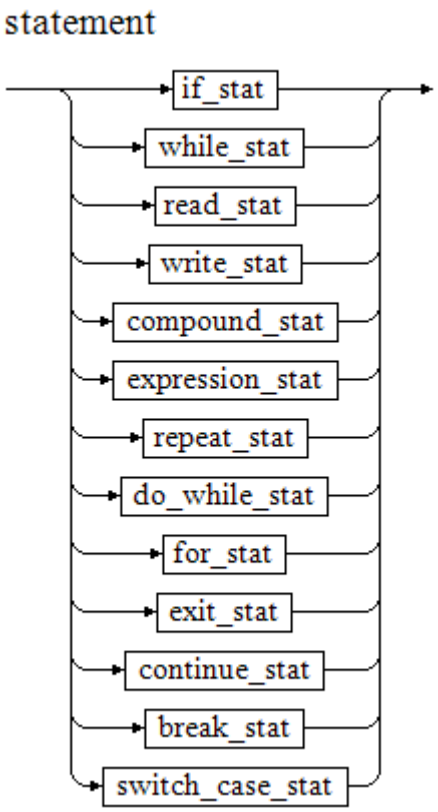
语法实现：

```
statement_list : statement_list statement
               |
```

规则 11：

```
statement = if_stat | while_stat | read_stat | write_stat | compound_stat |
expression_stat | repeat_stat | do_while_stat | for_stat | exit_stat |
continue_stat | break_stat | switch_case_stat.
```

语法图：



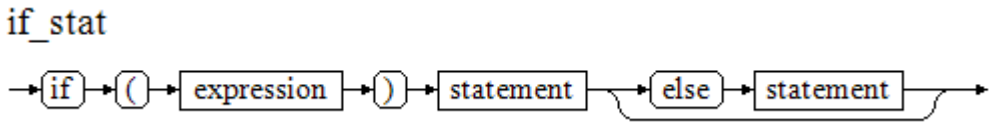
语法实现：

```
statement : if_stat
          | while_stat
          | read_stat
          | write_stat
          | compound_stat
          | expression_stat
          | repeat_stat
          | do_while_stat
          | for_stat
          | exit_stat
          | continue_stat
          | break_stat
          | switch_case_stat
```

规则 12：

```
if_stat = "if" "(" expression ")" statement ["else" statement].
```

语法图：



语法实现：

```
if_stat : if LPAREN expression RPAREN gen_jpc statement
        | if LPAREN expression RPAREN gen_jpc statement else gen_jpc_back
gen_jmp statement
```

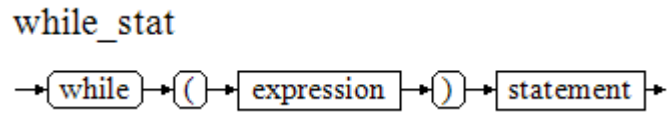
说明：

gen_jpc 用于在 if 语句判断条件之后，生成一条待回填的 jpc 指令；gen_jpc_back 用于回填上述判断条件之后的 jpc 指令；gen_jmp 此处用于生成跳过 else 语句的 jmp 指令。

规则 13：

```
while_stat = "while" "(" expression ")" statement .
```

语法图：



语法实现：

```
while_stat : while rec_loop_adr LPAREN expression RPAREN gen_jpc
statement
```

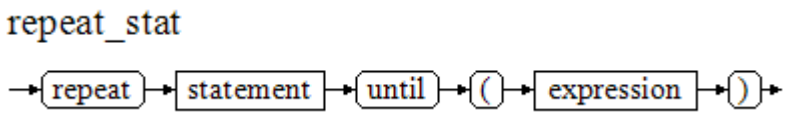
说明：

rec_loop_adr 用于记录循环的开始地址，此处记录 while 内判断条件的开始地址；gen_jpc 用于生成判断是否跳出循环的 jpc 指令。

规则 14：

```
repeat_stat = "repeat" statement "until" "(" expression ")".
```

语法图：



语法实现：

```
repeat_stat : repeat rec_loop_adr statement until LPAREN expression
RPAREN
```

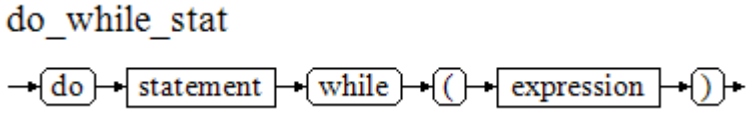
说明：

`rec_loop_adr` 此处用于记录 `repeat` 的开始地址。

规则 15：

```
do_while_stat = "do" statement "while" "(" expression ")".
```

语法图：



语法实现：

```
do_while_stat : do rec_loop_adr statement while LPAREN expression RPAREN
```

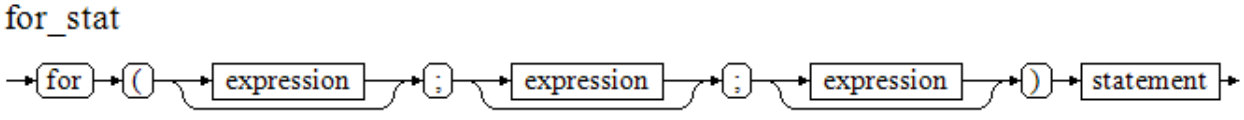
说明：

`rec_loop_adr` 此处用于记录 `do while` 中循环体的开始地址。

规则 16：

```
for_stat = "for" "(" [expression] ";" [expression] ";" [expression] ")" statement.
```

语法图：



语法实现：

```
for_stat : for LPAREN expression SEMICOLON rec_loop_adr expression
SEMICOLON gen_jpc gen_jmp rec_ex3_adr expression RPAREN gen_jmp_condition
gen_jmp_back statement
         | for LPAREN SEMICOLON rec_loop_adr expression SEMICOLON gen_jpc
gen_jmp rec_ex3_adr expression RPAREN gen_jmp_condition gen_jmp_back
statement
         | for LPAREN expression SEMICOLON rec_loop_adr SEMICOLON gen_jpc
gen_jmp rec_ex3_adr expression RPAREN gen_jmp_condition gen_jmp_back
statement
         | for LPAREN expression SEMICOLON rec_loop_adr expression
SEMICOLON gen_jpc gen_jmp rec_ex3_adr RPAREN gen_jmp_condition
gen_jmp_back statement
         | for LPAREN SEMICOLON rec_loop_adr SEMICOLON gen_jpc gen_jmp
rec_ex3_adr expression RPAREN gen_jmp_condition gen_jmp_back statement
         | for LPAREN SEMICOLON rec_loop_adr expression SEMICOLON gen_jpc
gen_jmp rec_ex3_adr RPAREN gen_jmp_condition gen_jmp_back statement
         | for LPAREN expression SEMICOLON rec_loop_adr SEMICOLON gen_jpc
gen_jmp rec_ex3_adr RPAREN gen_jmp_condition gen_jmp_back statement
         | for LPAREN SEMICOLON rec_loop_adr SEMICOLON gen_jpc gen_jmp
rec_ex3_adr RPAREN gen_jmp_condition gen_jmp_back statement
```

说明：

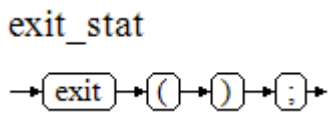
rec_loop_adr 用于记录 for 循环判断条件起始地址；gen_jpc 用于生成跳出循环体的 jpc 指令；gen_jmp 用于生成跳转到循环体内部的 jmp 指令；rec_ex3_adr 用于记录 for 循环第三个表达式的开始地址，便于每次结束循环后跳转执行第三个表达式；gen_jmp_condition 用于生成跳转回到 for 循环起始判断条件地址的 jmp 指令；gen_jmp_back 用于回填跳转到循环体内部的 jmp 指令。

【注意】for 循环的三个表达式可能存在缺失，可能有一个或两个缺失，甚至全部缺失。

规则 17：

```
exit_stat = "exit" "(" ")" ";".
```

语法图：



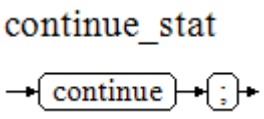
语法实现：

```
exit_stat : exit LPAREN RPAREN SEMICOLON
```

规则 18：

```
continue_stat = "continue" ";".
```

语法图：



语法实现：

```
continue_stat : continue SEMICOLON
```

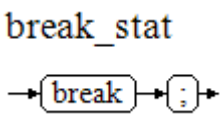
说明：

【注意】continue 语句只能用在循环里面，如果用在非循环部分，程序执行出错处理。

规则 19：

```
break_stat = "break" ";".
```

语法图：



语法实现：

```
break_stat : break SEMICOLON
```

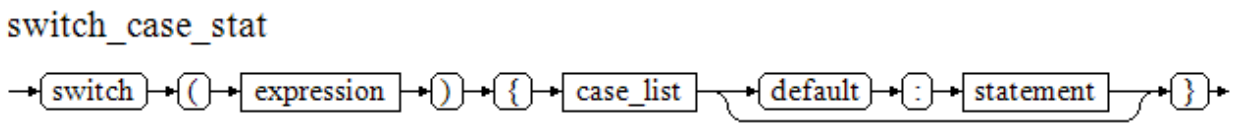
说明：

【注意】break 语句只能用在循环里面或是与 switch case 连用，如果用在非循环且非 switch case 部分，程序执行出错处理。

规则 20：

```
switch_case_stat = "switch" "(" expression ")" "{" case_list ["default" ":" statement] "}".
```

语法图：



语法实现：

```
switch_case_stat : switch LPAREN expression RPAREN LBRACE rec_switch_adr
case_list default COLON statement RBRACE
                 | switch LPAREN expression RPAREN LBRACE rec_switch_adr
case_list RBRACE
```

说明：

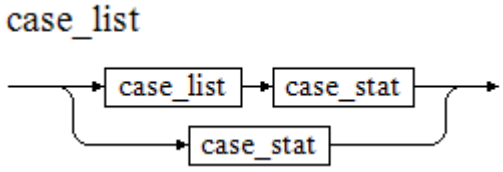
rec_switch_adr 用于存放 switch 分支结构的开始地址，以便之后 case 语句跳转到正确地址。

【注意】default 语句为可选语句，default 语句不含 break。

规则 21：

```
case_list = case_list case_stat | case_stat.
```

语法图：



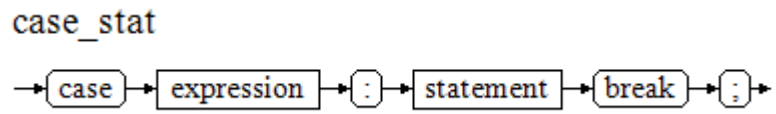
语法实现：

```
case_list : case_list case_stat
          | case_stat
```

规则 22：

```
case_stat = "case" expression ":" statement "break" ";".
```

语法图：



语法实现：

```
case_stat : case expression COLON gen_opr_switch gen_jpc statement break
SEMICOLON
```

说明：

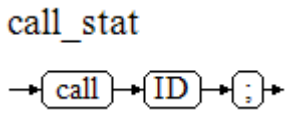
gen_opr_switch 用于根据 switch 分支结构的开始地址生成相应 opr 指令，gen_jpc 用于生成 jpc 条件判断指令，两者搭配便于跳转到下一条 case 语句或是跳出分支结构。

【注意】每条 case 语句必须在最后搭配一个 break。

规则 23：

```
call_stat = "call" ID ";".
```

语法图：



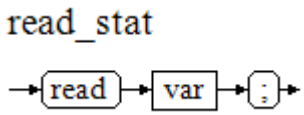
语法实现：

```
call_stat : call ID SEMICOLON
```

规则 24：

```
read_stat = "read" var ";".
```

语法图：



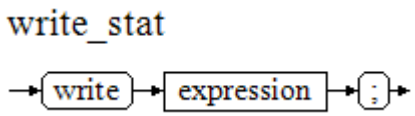
语法实现：

```
read_stat : read var SEMICOLON
```

规则 25：

```
write_stat = "write" expression ";".
```

语法图：



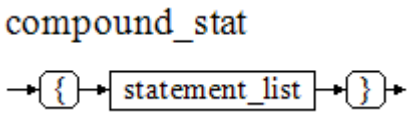
语法实现：

```
write_stat : write expression SEMICOLON
```

规则 26：

```
compound_stat = "{" statement_list "}".
```

语法图：



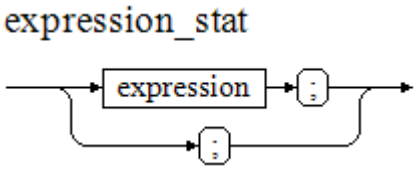
语法实现：

```
compound_stat : LBRACE statement_list RBRACE
```

规则 27：

```
expression_stat = expression ";" | ";".
```

语法图：



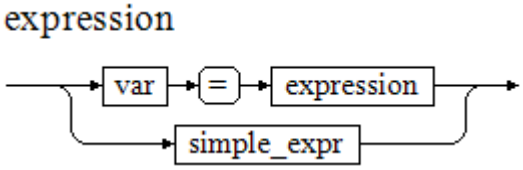
语法实现：

```
expression_stat : expression SEMICOLON
                | SEMICOLON
```

规则 28：

```
expression = var "=" expression | simple_expr.
```

语法图：



语法实现：

```
expression : var EQUAL expression
           | simple_expr
```

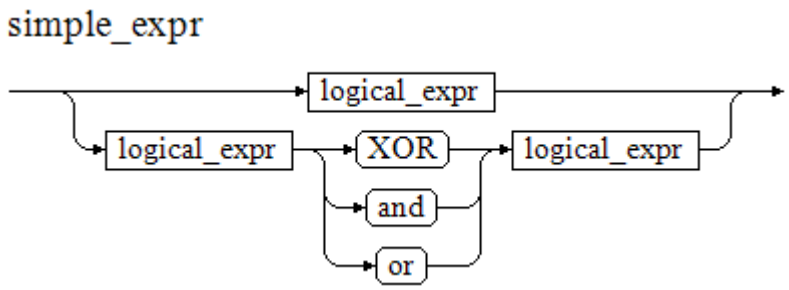
说明：

【注意】存在连续赋值的可能。

规则 29：

```
simple_expr = logical_expr | logical_expr ("XOR" | "and" | "or") logical_expr.
```

语法图：



语法实现：

```
simple_expr : logical_expr
            | logical_expr XOR logical_expr
            | logical_expr and logical_expr
            | logical_expr or logical_expr
```

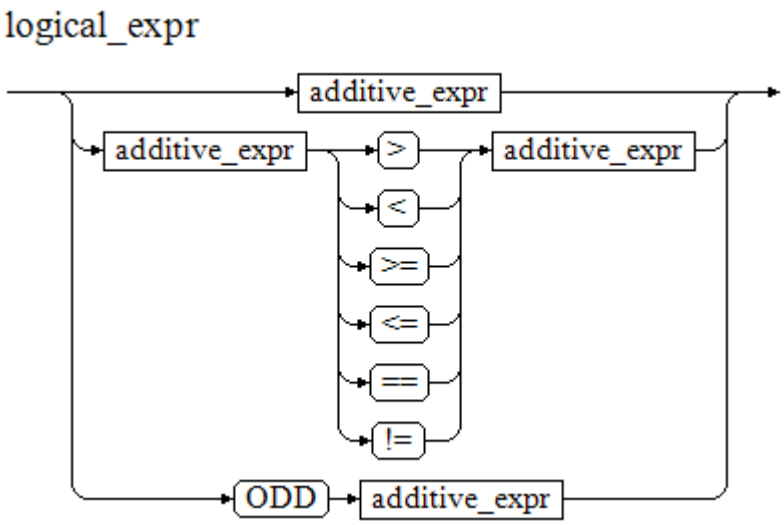
说明：

【注意】由于优先级的限制，logical_expr 可以不用加括号。

规则 30：

```
logical_expr = additive_expr | additive_expr (">" | "<" | ">=" | "<=" | "==" |
"!=") additive_expr | "ODD" additive_expr.
```

语法图：



语法实现：

```
logical_expr : additive_expr
             | additive_expr LSS additive_expr
             | additive_expr LEQ additive_expr
             | additive_expr GTR additive_expr
             | additive_expr GEQ additive_expr
             | additive_expr EQL additive_expr
             | additive_expr NEQ additive_expr
             | ODD additive_expr
```

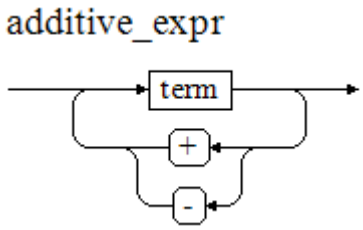
说明：

【注意】由于优先级的限制，additive_expr 可以不用加括号。

规则 31：

```
additive_expr = term {("+"|"-") term }.
```

语法图：



语法实现：

```
additive_expr : term
              | term PLUS additive_expr
              | term MINUS additive_expr
```

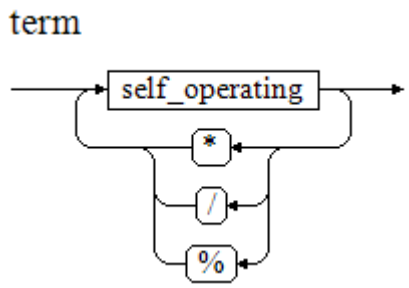
说明：

【注意】由于优先级的限制，term 可以不用加括号。

规则 32：

```
term = self_operating {("*"|"/"|"%") self_operating }.
```

语法图：



语法实现：

```
term : self_operating
     | self_operating TIMES term
     | self_operating DIVIDE term
     | self_operating MOD term
```

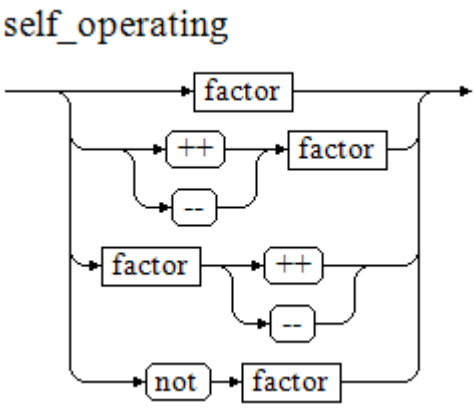
说明：

【注意】由于优先级的限制，self_operating 可以不用加括号。

规则 33：

```
self_operating = factor | ("++"|"--") factor | factor ("++"|"--") | "not" factor.
```

语法图：



语法实现：

```
self_operating : factor
               | SFPLUS factor
               | SFMINUS factor
               | factor SFPLUS
               | factor SFMINUS
               | not factor
```

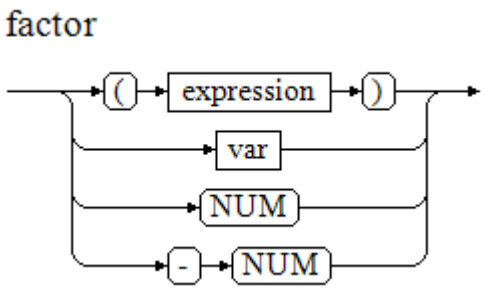
说明：

【注意】由于优先级的限制，factor 可以不用加括号。自增自减符号在前和在后的作用不同，如 x++的值为 x 的值，++x 的值为 x+1 的值。

规则 34：

```
factor = "(" expression ")"| var | NUM | "-" NUM.
```

语法图：



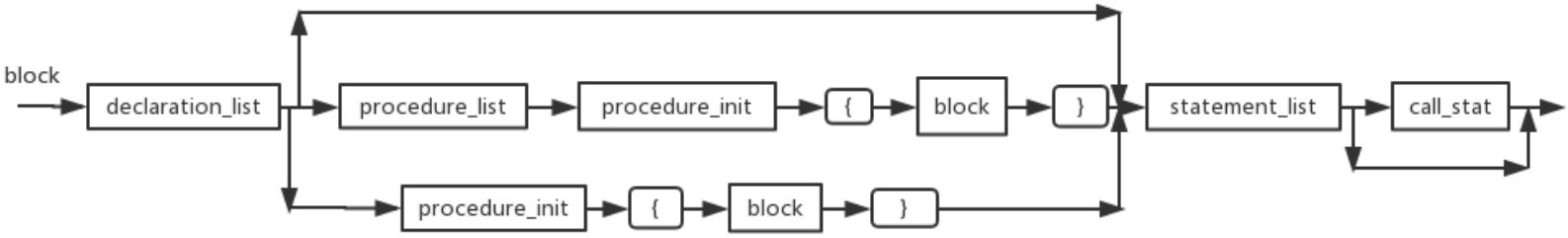
语法实现：

```
factor : LPAREN expression RPAREN
       | var
       | MINUS NUMBER
       | NUMBER
```

说明：

【注意】此处支持负整数。

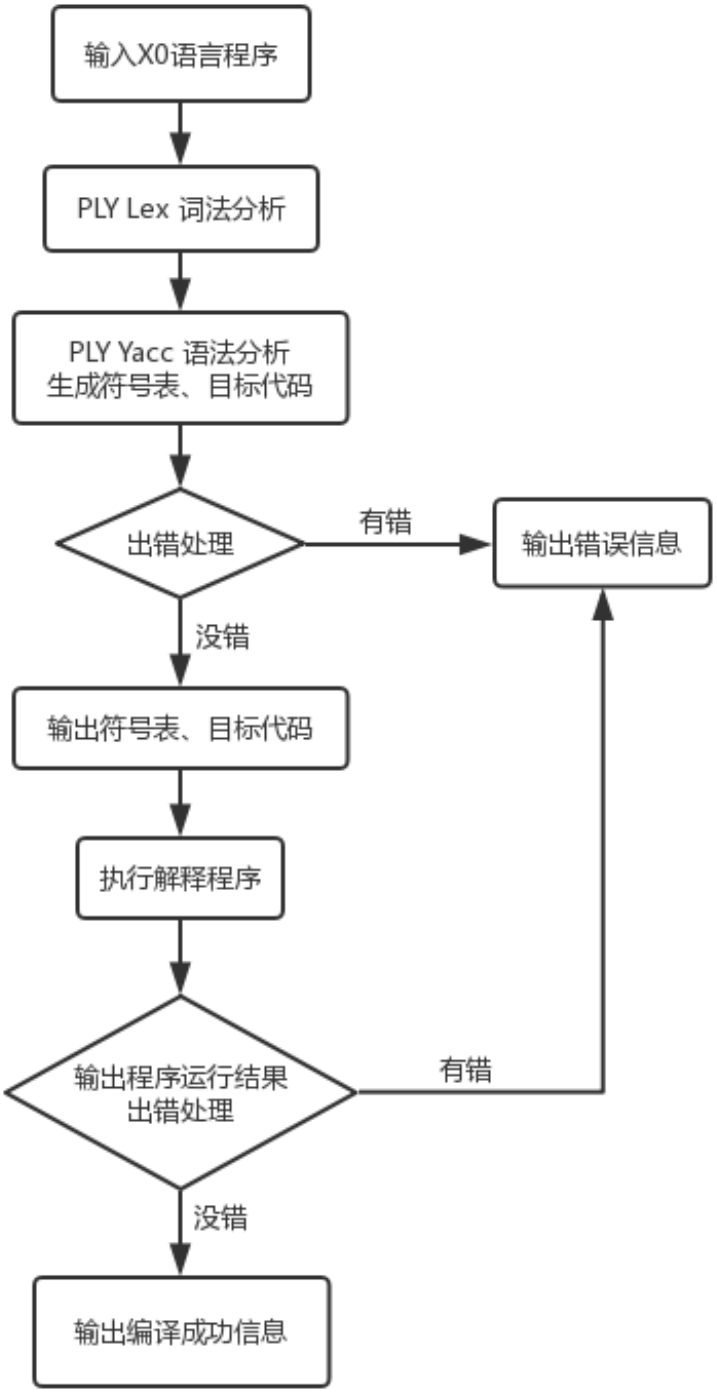
2.1.4 过程调用相关图



一个嵌套过程调用的例子：

```
main{
	const int m = 5;
	int a;
	int b;
	procedure g {
		procedure h {
			int i;
			int x;
			i = a;
			x = (m + i) / (m - i);
			if (x < (a + 1))
				b = x;
			write a;
			write i;
			write x;
			write b;
		}
		a++;
		call h;
	}
	a = 2;
	b = 1;
	call g;
}
```

2.1.5 编译程序总体结构



2.1.6 语法出错表定义

1) 过程嵌套层数超出上限

```
'Too many nesting levels!'
```

2) 标识符未定义

```
"ID '%s' undefined!" % p[1]
```

3) 在非循环语句中使用 continue

```
"'continue' is used in loops!"
```

4) 在非循环且非 switch case 语句中使用 break

```
"'break' is used in loops or the end of every case statement!"
```

5) 调用非过程的标识符

```
"ID '%s' is not procedure!" % p[1]
```

6) 数组越界

```
'Array index out of range!'
```

7) 数组元素未赋值就使用

```
'Array or variable not assigned!'
```

8) 标识符过多

```
'Too many identifiers!'
```

9) 生成的虚拟代码程序过长

```
'Program is too long!'
```

10) 地址偏移越界

```
'Displacement address is too big!'
```

11) int、char、bool 类型赋值越界

```
"The range of type 'int' is -2147483648 ~ 2147483647!"
"The range of type 'char' is -128 ~ 127!"
"The range of type 'bool' is 0 ~ 1!"
```

12) int、char、bool 类型输入值类型错误

```
"The type of input should be 'int'!"
"The type of input should be 'char'!"
"The type of input should be 'bool'!\nPlease input 'true' or 'false'!"
```

13) 输入程序不符合 X0 语法规则

```
"Syntax error at '%s' at the line %d" % (p.value, p.lineno)
'Syntax error in input!'
```

2.2 虚拟机

2.2.1 虚拟机组织结构

程序存储器	指令寄存器	数据寄存器	程序地址寄存器	基本地址寄存器	地址寄存器
instruction	current_code	s	p	b	t

程序存储器 instruction 用来存放通过编译产生的中间代码程序（目标程序），它在程序解释执行过程中保持不变。

指令寄存器 current_code 用来存放正在解释执行的一条中间代码指令。

数据寄存器 s 即所用数据栈。

程序地址寄存器 p 含有下一条要从程序存储器取得的、被解释执行的指令的地址。

基本地址寄存器 b 为正在执行的过程段在数据栈的起始地址。

地址寄存器 t 用来存放数据栈 s 的栈顶位置。

2.2.2 虚拟机指令格式

instruction = [{'f': char, 'l': int, 'a': int},……]

程序存储器 instruction 是存放目标程序指令的指令列表，每条指令是一个含三对键值的字典。'f'是指令码，代表指令类型，对应的 value 值类型为 char，'l'和'a'是与指令码相关的参数，对应的 value 值类型为 int。

不同指令的'l'和'a'代表的含义不同：

f	l	a
ini		常量
lit		常量
lod	变量声明和使用时的层差	变量在过程段的偏移量
sto	层差	偏移量
cal	层差	被调用程序段在程序存储器里的起始地址
jmp		跳转的程序地址
jpc		跳转的程序地址
jnc		跳转的程序地址
opr		运算类别

2.2.3 虚拟机指令系统及其解释

ini 0,a	预留 a 个存储位置
lit 0,a	将常数 a 置入栈顶
lod l,a	将 l,a 形成的栈地址变量值置入栈顶
sto l,a	将栈顶值存到由 l,a 形成的栈地址变量
cal l,a	调用一个子程序
jmp 0,a	无条件转移，直接跳转到相应程序段
jpc 0,a	有条件转移，若满足上述判断条件，则不跳转；不满足，则跳转
jnc 0,a	有条件转移，若满足上述判断条件，则跳转；不满足，则不跳转
opr 0,0	返回调用程序
opr 0,1	取负
opr 0,2	相加
opr 0,3	相减
opr 0,4	相乘
opr 0,5	相除
opr 0,6	判断奇偶
opr 0,8	判断是否相等
opr 0,9	判断是否不相等
opr 0,10	判断是否小于
opr 0,11	判断是否大于等于
opr 0,12	判断是否大于
opr 0,13	判断是否小于等于
opr 0,14	将栈顶值输出
opr 0,15	输出换行符
opr 0,16	读入一个输入置于栈顶
opr 0,17	自增，操作符在前，即++x 为原 x+1 的值，x 自增 1
opr 0,18	自减，操作符在前，即--x 为原 x-1 的值，x 自减 1
opr 0,19	自增，操作符在后，即 x++为原 x 的值，x 自增 1
opr 0,20	自减，操作符在后，即 x--为原 x 的值，x 自减 1

opr 0,21	取余
opr 0,22	异或，若逻辑两边相等，则返回 0；不相等，则返回 1
opr 0,23	switch case 语句辅助，若满足 case 条件，则 pop 栈顶值，并将新的栈顶值置为 1，便于跳出分支结构语句；否则，将栈顶值置为 0，便于进入下一个 case 语句判断
opr 0,24	逻辑与，若逻辑两边均不为 0，则返回 1；否则，返回 0
opr 0,25	逻辑或，若逻辑两边均为 0，则返回 0；否则，返回 1
opr 0,26	逻辑非，若表达式为 0，则返回 1；否则，返回 0

lod 和 sto 指令的多种情况：

1) lod 或 sto 一个变量：参数 l 为层差，参数 a 为该变量在过程段的偏移量。
2) lod 或 sto 一个索引可知的数组变量（索引可知意味着索引可以直接通过计算得到，上下文无关）：参数 l 为层差，参数 a 为计算得到的该索引数组的偏移量。
3) lod 或 sto 一个索引未知的数组变量（索引未知意味着索引值必须通过上文得到的结果才能计算得到，上下文有关）：参数 l 为层差，参数 a 为该数组变量的起始偏移量的负值。

3. 程序模块架构及运行流程

本项目程序分为三部分，都存放在 code 文件夹中：lex_analysis.py，compiler.py，x0_define_exe.py。

lex_analysis.py 用于构建 lex 词法分析器。

compiler.py 获取源程序之后，import lex_analysis 构建好的 lex 词法分析器，得到 tokens，再使用 PLY 库中的 Yacc 进行语法分析，并同时编写函数代码，生成符号表信息和目标程序指令。

在 code 文件夹同级目录下还有一个 result_files 文件夹，在该文件夹下创建 foutput.txt、ftable.txt、fcode.txt 文件。

在上述翻译过程中，同时进行着出错信息的处理和记录，完成翻译过程后，若存在出错信息，则在 foutput 文件中记录错误信息数量和出错信息内容。

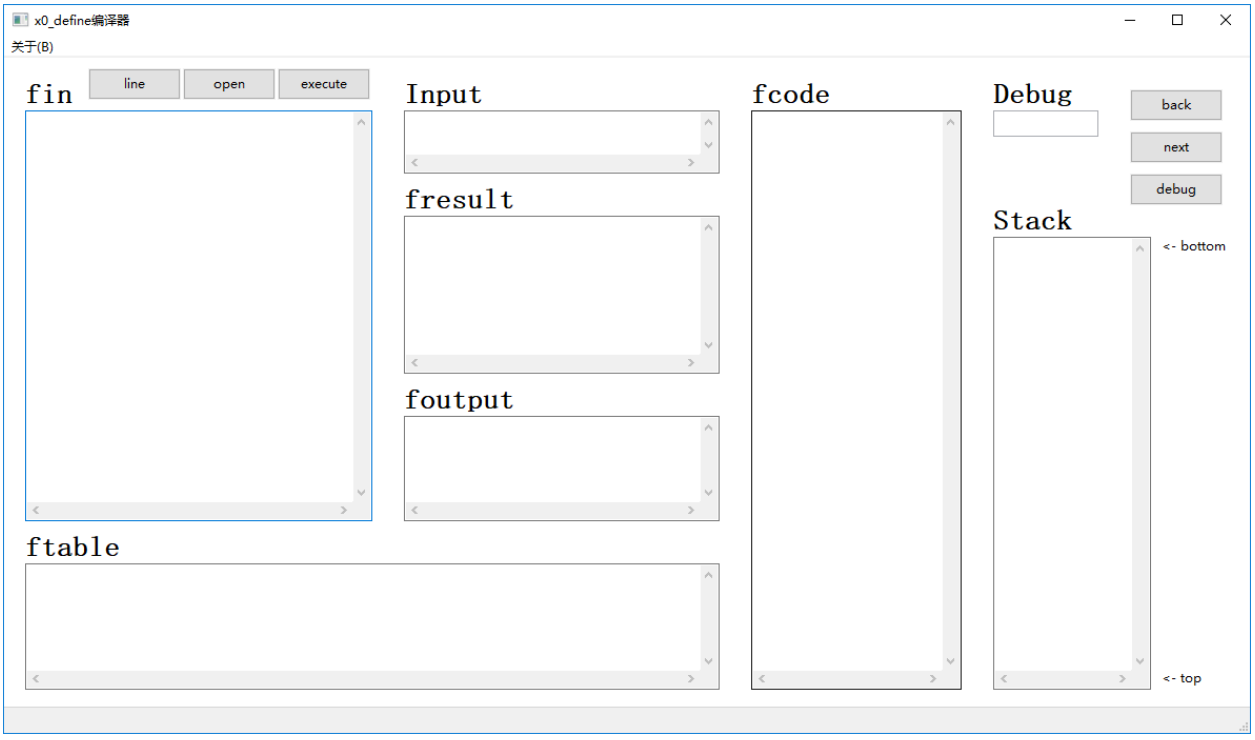
若没有出错信息，则将生成的符号表信息存入 ftable 文件，将目标程序指令代码存入 fcode 文件，再进行解释程序。

在 result_files 文件夹下创建 fresult.txt 文件，解释程序一边解析指令代码，一边将相应的变量输入和结果输出存入 fresult 文件，同时进行出错信息的处理和记录。

完成解释程序后，若存在出错信息，则在 foutput 文件中记录错误信息数量和出错信息内容；若没有出错信息，则在 foutput 文件中写入运行成功信息“Parsing success!”。

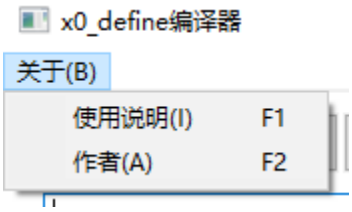
x0_define_exe.py 则负责构建前端 UI 界面，提供友好的用户接口，用户可以在前端手动输入或者打开 X0 语言程序，输入可能需要的输入值，执行之后就能看到相应的符号表 ftable、指令代码 fcode、输出信息 foutput、运行结果 fresult。

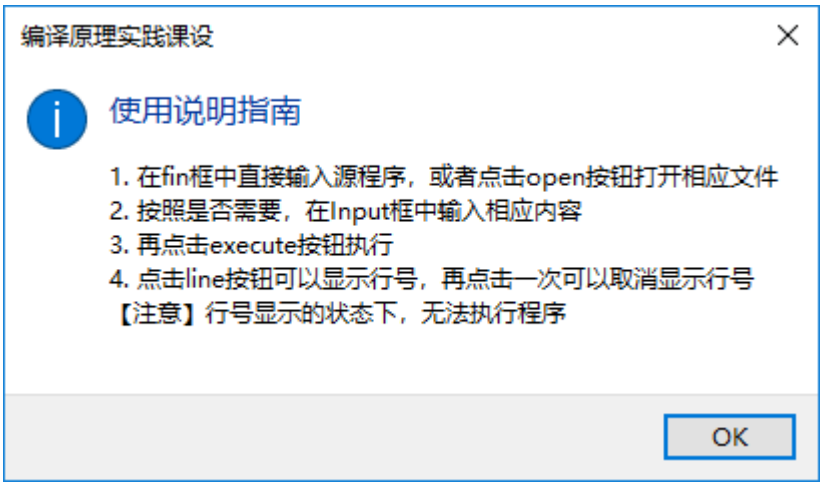
4. 前端 UI 页面介绍



4.1 菜单栏

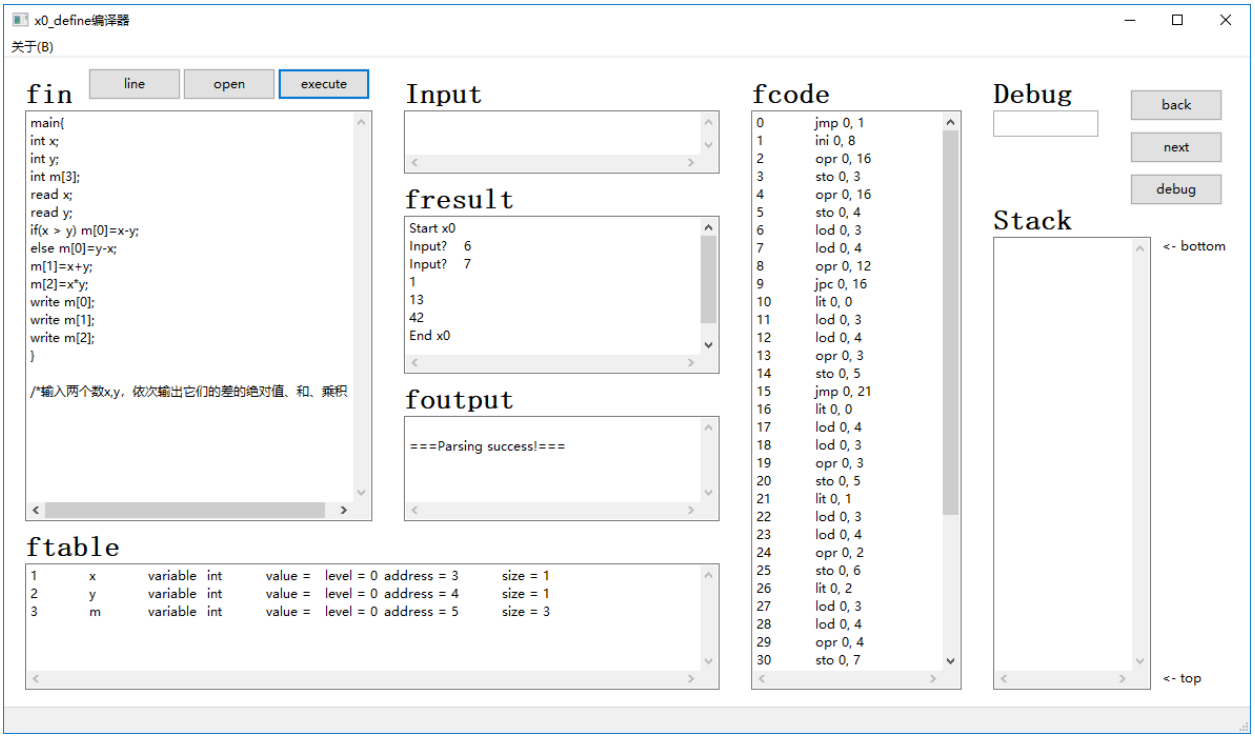
在菜单栏中可以查看使用说明和作者信息。





4.2 编译器使用说明

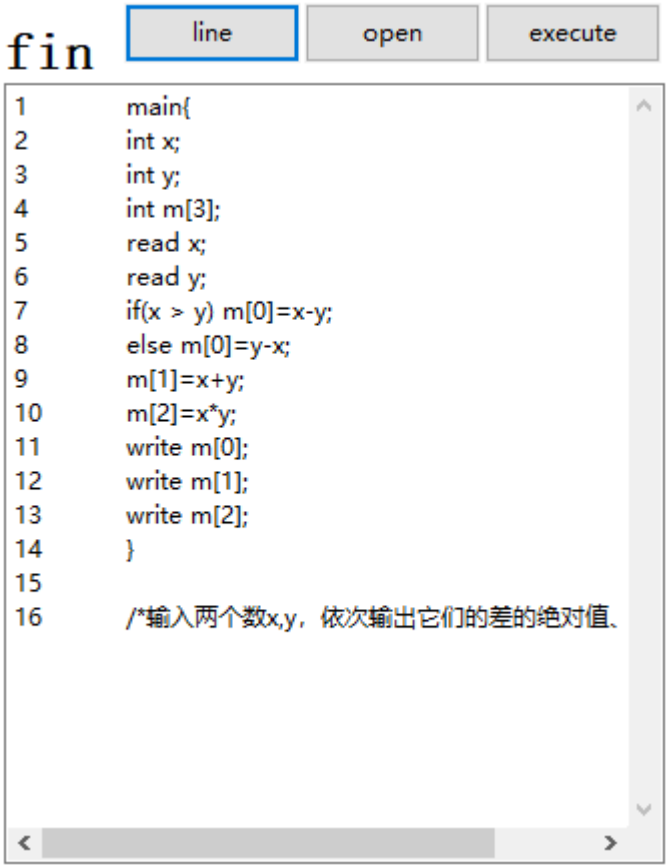
1) 在 fin 框中直接输入源程序，或者点击 open 按钮打开相应文件，文件中的内容就会显示在 fin 框中
2) 如果该程序需要输入变量，则在 Input 框中输入相应内容和数量的输入信息
3) 点击 execute 按钮执行
4) 在 ftable、fresult、foutput、fcode 框中就能看到相应的内容，Input 框则被清空



4.3 行号功能

点击 line 按钮，fin 框中的源程序就会显示相应行号，再点击一次，可以取消显示行号。

【注意】在行号显示开启的状态下，无法正常执行程序，想要执行程序需要取消行号显示



4.4 Debug 目标代码单步执行查看数据栈

行号显示取消功能本来是为了方便 Debug 才实现的，原先的设想是：想要 Debug 必须开启行号显示，debug 过程中 ftable 框、foutput 框、fcode 框都不变化。在 Debug 框中输入断点行号（想要程序运行停在哪一行之前），如果需要输入变量的话还要在 Input 框中输入相应内容，点击 debug 按钮，编译器就会到解析程序中获取当前状态下的数据栈信息，并显示在 Stack 框中。之后可以重新输入断点行号，点击 debug 按钮，也可以直接点击 back 或者 next 按钮，获取上一步或下一步的数据栈信息并显示。

但在实现过程遇到了不少困难，程序需要知道每一行源程序对应到目标指令代码的哪个部分，才能够获取这个状态下的数据栈信息，因此我想要获取指令代码对应的源程序行号，但我尝试了多种方法，都没能成功得到想要的行号。以及在上一步和下一步的过程中，可能出现指令跳转循环，此时便不能单纯通过行号加 1 或减 1 来判断数据栈的内容。

因此最后还是没能实现该功能，但是保留了行号功能和前端的页面显示（只是一个显示、没有实际效果）。

5. 程序模块前后端接口

上述内容中已经说明过，compiler.py 程序将输出 ftable.txt、fcode.txt、fresult.txt、foutput.txt 这四个文件。

在前端页面输入源程序和需要输入的变量信息后，将源程序和输入变量列表一并传给后端。

```
argv_list = self.input.GetValue().strip('\n').split('\n')
compiler.x0_compiler(self.fin.GetValue(), argv_list)
```

在后端程序中，parameter 变量为当前从命令行读取的参数索引，argv_list 列表为从命令行读取的参数列表，将前端的 argv_list 列表输入信息 copy 到后端的 argv_list 列表，模拟为从命令行获取的参数列表。

```
def x0_compiler(data, input_list):
   parameter = 0
   argv_list = input_list.copy()
   ……
```

这样就可以实现前后端连接，之后前端 ftable 框、fcode 框、fresult 框、foutput 框只需要直接从后端生成的四个文件中读取即可。

6. 后端全局数据结构、常量和变量

全局常量：

```
table_max_num = 100  # 符号表容量
address_max = 2048  # 地址上界
code_max_num = 200  # 最多的虚拟机代码数
stack_max_num = 700  # 运行时数据栈元素最大数量
lev_max_num = 3  # 最大允许过程嵌套声明层数
```

全局变量：

```
cx = 0  # 虚拟机代码索引
tx = 0  # 符号表当前尾指针
lev = 0  # 层次记录
parameter = 0  # 当前从命令行读取的参数索引
local_address = 3  # 局部变量地址
```

全局数据结构：

```
table = []  # table = [{'name': ID, 'kind': ,'type': , 'value': , 'level': , 'address': , 'size': }] 符号表
instruction = []  # instruction = [{'f': char, 'l': int, 'a': int}] 存放虚拟机代码
argv_list = []  # 从命令行读取的参数列表
local_address_list = []  # 存放每层的局部变量地址
error_list = []  # 出错列表
loop_adr_list = []  # 存放每次循环语句循环的开始地址
ex3_adr_list = []  # 存放 for 循环表达式 3 的开始地址
for_start_adr = []  # 存放 for 循环的开始地址
isbreak_list = []  # 是否在循环中使用了 break 语句，1 为是，空为否，每层增加一个元素
switch_start_adr = []  # 存放 switch 分支结构的开始地址
```

7. 后端函数原型

函数原型	p_program(p) p_block(p) p_rec_local_adr(p) p_procedure_list(p) p_procedure_stat(p) p_procedure_init(p) p_gen_ini(p) p_declaration_list(p) p_declaration_stat(p) p_type(p) p_var(p) p_statement_list(p) p_statement(p) p_gen_jpc_back(p) p_gen_jpc(p) p_gen_jmp(p) p_if_stat(p) p_rec_loop_adr(p) p_while_stat(p) p_repeat_stat(p) p_do_while_stat(p) p_rec_ex3_adr(p) p_gen_jmp_back(p) p_gen_jmp_condition(p) p_for_stat(p) p_exit_stat(p) p_continue_stat(p) p_break_stat(p) p_rec_switch_adr(p) p_switch_case_stat(p) p_case_list(p) p_gen_opr_switch(p) p_case_stat(p) p_call_stat(p) p_read_stat(p) p_write_stat(p) p_compound_stat(p) p_expression_stat(p) p_expression(p) p_simple_expr(p) p_logical_expr(p) p_additive_expr(p) p_term(p) p_self_operating(p) p_factor(p) p_error(p)
参数描述	p: PLY Yacc 中包含了组成当前规则的所有语法符号的序列
函数描述	PLY Yacc 用于语法分析、出错处理、辅助生成目标指令代码的函数
返回值	无

函数原型	enter(name, kind, type, value, level, address, size)
参数描述	name: 标识符 ID 名 kind: 标识符类型，常量 constant、变量 variable 或过程 procedure type: 常量或变量的标识符类型，整型 int、字符型 char 或布尔型 bool，过程标识符的 type 类型置为空 value: 标识符的值，常量正常记录其值，变量和过程的值记录为-1 level: 标识符的层数，变量和过程正常记录其层数，常量的层数记录为-1 address: 标识符在本层的偏移量，变量正常记录其局部地址，过程标识符的地址为其在指令代码中的程序地址，常量的地址记录为-1

	size: 标识符的存储空间大小，常量的大小记录为-1，单个变量的大小为 1，数组变量的大小为数组开辟空间的大小，过程标识符的大小为该过程初始化之后的局部变量地址，即局部变量个数加 3
函数描述	将声明的常量、变量、过程存入符号表中
返回值	无

函数原型	gen(f, l, a)
参数描述	f: 指令码，代表指令类型 l: 与指令码相关的参数 a: 与指令码相关的参数 （具体参数描述见 2.2.2）
函数描述	生成虚拟机指令代码并存入程序存储器
返回值	无

函数原型	base(l, s, b)
参数描述	l: 层差 s: 数据栈 b: 指令基址
函数描述	通过过程基址求上 l 层过程的基址
返回值	上 l 层过程的基址

函数原型	interpret()
参数描述	无
函数描述	解释程序，解释执行指令代码并得到 fresult 运行结果文件
返回值	无

函数原型	x0_compiler(data, input_list)
参数描述	data: 源程序代码 input_list: 命令行输入组成的字符列表

函数描述	构建 yacc 语法分析器，与 lex 词法分析器一起翻译源程序，得到 ftable 符号表文件和 fcode 指令代码文件，再调用解释程序解释执行目标指令代码，同时上述过程中进行出错处理，得到 foutput 输出信息文件
返回值	无

8. 编译器主要功能实现算法

8.1 负数的输入支持

Factor 中增添负数（即 MINUS NUMBER）的情况。

8.2 多类型支持

在 type 类型函数中增加 bool 类型，各种变量输入值和输出值都要根据变量具体的 type 类型进行判断，尝试将输入的值转换为编译器内部计算使用的整型数值，或者尝试将输出的整型数值转换为相应变量原本的类型，若尝试失败，则进入出错处理。

8.3 常量、变量与过程

常量、变量的声明与过程的声明区分开来，放在不同的函数中，常量声明需要以 const 开头且声明的同时赋初值。符号表中，除了 name、kind 是三种类型声明都会有的字段，其他字段则根据不同的声明而有所不同。

常量声明时，type 字段为相应常量类型，value 字段为赋予的初值，level、address、size 字段置为-1。

变量声明时，type 字段为相应变量类型，value 字段置为-1，level 和 address 字段则为层数和局部偏移地址，size 字段根据变量的类型来赋值，单个变量赋值为 1，数组变量赋值为数组的大小。

过程声明时，type 字段置为空，value 字段置为-1，address 和 size 字段置为初始值 1 和 0，之后待回填。

8.4 if 语句的实现

对于无 else 的 if 语句，在条件判断结束之后，生成待回填的 jpc 指令，用于判断是否跳转，在语句解析结束后回填 jpc 指令。

对于有 else 的 if else 语句，在条件判断结束之后，同样生成待回填的 jpc 指令；在 if 部分语句解析结束后，回填 jpc 指令，并生成待回填的 jmp 跳转指令，用于跳转到 else 语句末尾；在 else 部分语句解析结束后，回填 jmp 指令。

8.5 while 语句的实现

在语句开始时，记录循环初始地址，便于后面指令的跳转；解析完 while 的判断条件后，生成待回填的 jpc 指令，用于跳出循环；在循环体语句解析结束后，生成 jmp 指令，跳转回到循环起始地址，并回填 jpc 指令，最后移除循环初始地址列表的栈顶值。

8.6 repeat until 语句的实现

在语句开始时，记录循环初始地址，便于后面指令的跳转；解析完循环体语句和判断条件后，根据循环初始地址生成 jpc 跳转指令，用于跳出循环或者跳回到循环开始位置，最后移除循环初始地址列表的栈顶值。

8.7 do while 语句的实现

在语句开始时，记录循环初始地址，便于后面指令的跳转；解析完循环体语句和判断条件后，根据循环初始地址生成 jnc 跳转指令，用于跳出循环或者跳回到循环开始位置，最后移除循环初始地址列表的栈顶值。（jnc 指令跟 jpc 指令正好相反：当条件满足时，jpc 不跳转，jnc 跳转；当条件不满足时，jpc 跳转，jnc 不跳转）

8.8 for 语句的实现

注意 for 语句存在括号内表达式 1、表达式 2、表达式 3 中任意数量表达式缺失的情况。

解析完表达式 1 后，记录循环初始地址，便于后面指令的跳转；解析完表达式 2 后，生产待回填的 jpc 指令和 jmp 指令，jpc 指令用于跳出循环或者进入循环，jmp 指令用于进入循环后跳过表达式 3 的位置，同时记录表达式 3 的起始位置；解析完表达式 3 后，生成跳转指令，跳转到循环开始位置，并将循环初始地址剪切到新的变量列表中，这是为了使 break 语句和 continue 语句能够正常运

作的同时，防止 for 循环和其他循环嵌套时出错；回填 jmp 指令、jpc 指令，并生成 jmp 指令，用于跳转到表达式 3 的起始位置。

for 语句解析结束后，移除表达式 3 起始位置列表的栈顶值，移除 for 循环起始地址列表的栈顶值。

8.9 exit 语句的实现

生成 opr 结束运行指令。

8.10 continue 语句的实现

continue 语句只能用在循环中，不能用在其他地方。首先需判断使用位置是否为循环内部，若为循环内部，则根据循环是 for 循环还是其他循环，生成相应的 jmp 指令，用于跳转到循环起始位置。

8.11 break 语句的实现

break 语句只能用在循环中、或者与 switch case 语句搭配使用，不能用在其他地方。首先需判断使用位置是否为循环内部，若为循环内部，生成待回填的 jmp 指令，用于跳转出循环，并在 isbreak_list 列表中添加一条记录，用于记录当前循环内是否含 break 语句。

对于 while、repeat until、do while 循环中的 break 语句：while 循环结束解析结束后，从当前指令的前一条开始，倒序查找是否有待回填的 jmp 指令（即 break 语句生成的 jmp 指令），直到查找到当前循环的开始位置；对于查找到的待回填的 jmp 指令，进行回填；最后移除 isbreak_list 列表的栈顶值，移除循环起始地址列表的栈顶值。

对于 for 循环中的 break 语句，操作与其他循环类似，只是将循环起始地址列表改为 for 循环专属的循环起始地址列表。

8.12 switch case 语句的实现

注意 case 语句一定要在末尾搭配 break 语句。可以没有 default 语句。

解析完 switch 的表达式后，记录分支结构起始地址，便于后面指令的跳转。

在 case 部分语句中，解析完 case 的表达式后，生成 opr 自定义操作指令，用于进行合适的跳转（若两者表达式相等，移除数据栈栈顶值，并将新的栈顶值置为 1，以便进入该 case 语句内部；若不相等，将栈顶值置为 0，以便跳转到下一个 case 语句处）；生成待回填的 jpc 指令，用于进入

case 语句内部或跳转到下一个 case 语句处；case 语句解析结束后，回填上一个 jpc 指令，并生成待回填的 jmp 指令，用于跳转出分支结构。

解析完 switch case 语句后，从当前指令的前一条开始，倒序查找是否有待回填的 jmp 指令（即 case 语句生成的 jmp 指令），直到查找到当前分支结构的开始位置；对于查找到的待回填的 jmp 指令，进行回填，最后移除分支结构起始地址列表的栈顶值。

8.13 call 语句的实现

从符号表中查找想要调用的过程的层号和局部地址，以此计算层差并生成 cal 指令。

注意标识符未定义、和调用了非过程标识符的出错处理。

8.14 read 语句的实现

read 语句一次只能读入一个值。

如果要将读入的值赋给单个变量，则直接按照这个变量在符号表中的层级和局部地址来生成 sto 指令；如果要将读入的值赋给单个数组变量，则相应 sto 指令的地址字段还需加上数组索引。

8.15 write 语句的实现

如果输出的是常数、表达式、单个常量、单个变量，只需要直接把栈顶值输出；如果输出的是整个数组，则需要对整个数组进行挨个读入挨个输出。

8.16 过程调用

在每个 block 中，声明常量和变量之后，记录当前 local_address 局部偏移地址，存放到列表中，便于之后符号表中过程的 size 和指令代码中 jmp 指令的回填。再声明过程时，层数加 1，局部偏移地址初始化为 3。在解析 statement_list 中的内容时，lod、sto、cal 指令注意计算层级差。当过程声明及定义结构结束后，层数减 1，移除局部变量偏移地址的栈顶值。

8.17 表达式含有数组变量的情况

对于索引可知的数组变量（索引可知意味着索引可以直接通过计算得到，上下文无关），则 lod 和 sto 指令的参数 a 为计算得到的该索引数组的偏移量，读取数据栈中数据时，根据层差获得的基址和参数 a 索引直接计算得到绝对地址。

对于索引未知的数组变量（索引未知意味着索引值必须通过上文得到的结果才能计算得到，上下文有关），则 lod 和 sto 指令的参数 a 为该数组变量的起始偏移量的负值，读取数据栈中数据时，根据层差获得的基址、数据栈当前栈顶值直接计算得到绝对地址。

8.18 读取一个标识符的值

如果该标识符是常量，应生成 ini 指令；如果该标识符是变量，应生成 lod 指令。

8.19 运算符优先级

X0 扩展语言支持的运算符都较容易实现，此处不再赘述，需要注意的是各运算符之间的优先级，比如 3+4*5 是应该先执行 3+4 还是先执行 4*5。

使用 PLY Yacc 的 precedence 实现，使得上述这种情况不需要加括号就能判断先执行哪部分。

```
precedence = (
    ('left', 'and', 'or', 'XOR'),
    ('left', 'LSS', 'LEQ', 'GTR', 'GEQ', 'EQL', 'NEQ'),
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE', 'MOD'),
    ('right', 'SFPLUS', 'SFMINUS', 'not')
)
```

各运算符的优先级从低到高排序，同一个元组内运算符优先级相同，每个元组内第一个字段表示运算符结合方式（左结合、右结合、无结合方式）。

8.20 数组元素未赋值就使用

解释程序在开始运行时，就开辟了一块 stack_max_num 大小的列表空间，并给每个元素都赋值为-999，当读取数据栈内数据时，若数值为-999，则意味着该标识符的这个位置未赋初值，进行出错处理。

8.21 连续赋值

在 sto 指令的解释程序中，解释了当前 sto 指令后，向下一条指令扫描连接的 sto 指令，若存在连续的 sto 指令，则将栈顶值也赋给新的 sto 指令需要赋值的对象；当扫描到一条非 sto 指令时，视为连续赋值结束，跳出赋值语句。

附录 1：语法规则一览

x0.ebnf

```
program = "main" "{" block "}".
block = declaration_list procedure_list statement_list [call_stat].
procedure_list = [procedure_list procedure_stat | procedure_stat].
procedure_stat = procedure_init "{" block "}".
procedure_init = "procedure" ID.
declaration_list = [declaration_list declaration_stat | declaration_stat].
declaration_stat = type ID ";" | type ID "[" NUM "]" ";" | "const" type ID "="
["-"] NUM ";".
type = "int" | "char" | "bool".
var = ID | ID "[" expression "]".
statement_list = statement_list statement | "".
statement = if_stat | while_stat | read_stat | write_stat | compound_stat |
expression_stat | repeat_stat | do_while_stat | for_stat | exit_stat |
continue_stat | break_stat | switch_case_stat.
if_stat = "if" "(" expression ")" statement ["else" statement].
while_stat = "while" "(" expression ")" statement .
repeat_stat = "repeat" statement "until" "(" expression ")".
do_while_stat = "do" statement "while" "(" expression ")".
for_stat = "for" "(" [expression] ";" [expression] ";" [expression] ")" statement.
exit_stat = "exit" "(" ")" ";".
continue_stat = "continue" ";".
break_stat = "break" ";".
switch_case_stat = "switch" "(" expression ")" "{" case_list ["default" ":"
statement] "}".
case_list = case_list case_stat | case_stat.
case_stat = "case" expression ":" statement "break" ";".
call_stat = "call" ID ";".
read_stat = "read" var ";".
write_stat = "write" expression ";".
compound_stat = "{" statement_list "}".
expression_stat = expression ";" | ";".
expression = var "=" expression | simple_expr.
simple_expr = logical_expr | logical_expr ("XOR" | "and" | "or") logical_expr.
logical_expr = additive_expr | additive_expr (">" | "<" | ">=" | "<=" | "==" |
"!=") additive_expr | "ODD" additive_expr.
additive_expr = term {("+"|"-") term }.
term = self_operating {("*"|"/"|"%") self_operating }.
self_operating = factor | ("++"|"--") factor | factor ("++"|"--") | "not" factor.
factor = "(" expression ")"| var | NUM | "-" NUM.
```