

# CMM 解释器项目文档

## 目录

一、背景	3
二、文法描述 (BNF)	3
三、系统分析和设计	5
(一) 实验环境	5
(二) 系统架构设计	5
(三) 系统功能设计	6
四、功能模块详解	7
(一) 词法分析	7
1. 源代码读入	7
2. 词法单元识别	7
3. 字符读取	7
4. 错误处理	8
(二) 语法分析	9
1. 分析方法	9
2. 分析过程	9
3. 主要数据结构	9
4. 主要算法	10
(三) 语义分析	12
1. 符号表结构	12
2. 符号表构建	15
3. 语义分析	16
(四) 中间代码生成	18
1. 中间代码结构	19
2. 四元式类型	19
3. 特殊变量	20
4. 拉链反填实现	20
5. 举例分析	21
(五) 中间代码解释执行	23
1. 主要数据结构	23
2. 主要算法	24
(六) 界面设计	26
五、测试用例和测试结果	26
(一) 词法分析	27
(二) 语法分析	28
(三) 解释执行	31
(四) 界面	36
六、使用说明	36
七、小结	36

## 一、 背景

小组基本信息

姓名	学号	分工内容
@YieldNull	*****	全部模块

## 二、 文法描述 ( BNF )

```
NEWLINE ::= "\r"? "\n"
WS       ::= " |\t"

IF        ::= "if"
ELSE      ::= "else"
WHILE     ::= "while"
READ      ::= "read"
WRITE     ::= "write"
INT       ::= "int"
REAL      ::= "real"
RETURN    ::= "return"
VOID      ::= "void"
PLUS      ::= "+"
MINUS     ::= "-"
TIMES     ::= "*"
DIVIDE    ::= "/"
ASSIGN    ::= "="

LT        ::= "<"
GT        ::= ">"
EQUAL     ::= "=="
NEQUAL    ::= "<>"

LPAREN    ::= "("
RPAREN    ::= ")"
LBRACE    ::= "{"
RBRACE    ::= "}"
LBRACKET  ::= "["
RBRACKET  ::= "]"

ROWCOMM   ::= "//"
MULCOMM   ::= "/* (~[\"*\"]) * \"*\" (~[\"/\"]) (~[\"*\"]) * \"*\" */"
COMMA     ::= ","
SEMICOLON ::= ";"
```

LETTER	::=	["a"-"z"]["A"-"Z"]
DIGIT	::=	["0"-"9"]
ID	::=	<LETTER> ( ( <LETTER>   <DIGIT>   "_" ) * ( <LETTER>   <DIGIT> ) )?
INT_LITERAL	::=	["1"-"9"] (<DIGIT>)*   "0"
REAL_LITERAL	::=	<INT_LITERAL> ( "." ( <INT_LITERAL> )+ )?
program	::=	exterStmts
exterStmts	::=	( declareStmt   FuncDefStmt )*
innerStmts	::=	( ifStmt   whileStmt   declareStmt   assignStmt   funcCallStmt   returnStmt )*
funcDefStmt	::=	returnType <ID> <LPAREN> ( FuncDefParamList )? <RPAREN> <LBRACE> innerStmts <RBRACE>
funcDefParam	::=	dataType <ID>
funcDefParamList	::=	funcDefParam ( <COMMA> funcDefParam )*   <VOID>
funcCallExpr	::=	<ID> <LPAREN> ( funcCallParamList )? <RPAREN>
funcCallStmt	::=	funcCallExpr <SEMICOLON>
funcCallParamList	::=	expr ( <COMMA> expr )*   <VOID>
funcType	::=	<VOID>   dataType
returnStmt	::=	<RETURN> expression <SEMICOLON>
dataType	::=	<INT>   <REAL>
array	::=	<LBRACKET> ( <INT_LITERAL>   <ID> )? <RBRACKET>
arrayInit	::=	<LBRACE> ( INT_LITERAL ( <COMMA> INT_LITERAL )*   REAL_LITERAL ( <COMMA> REAL_LITERAL )* ) <RBRACE>
declareStmt	::=	dataType (array)? <ID> ( <COMMA> <ID> )* ( <ASSIGN> expression   arrayInit ) <SEMICOLON>
assignStmt	::=	<ID> (array)? <ASSIGN> expression <SEMICOLON>
ifStmt	::=	<IF> <LPAREN> condition <RPAREN> <LBRACE> innerStmts <RBRACE> ( <ELSE> <LBRACE> innerStmts <RBRACE> )?
whileStmt	::=	<WHILE> <LPAREN> condition <RPAREN> <LBRACE> innerStmts <RBRACE>
condition	::=	expression compOp expression
expression	::=	term (addOp term)*
term	::=	factor (mulOp factor)*
factor	::=	<REAL_LITERAL>   <INT_LITERAL>   <ID> ( array )?   funcCallExpr   <LPAREN> expression <RPAREN>
compOp	::=	<LT>   <GT>   <EQUAL>   <NEQUAL>
addOp	::=	<PLUS>   <MINUS>
mulOp	::=	<TIMES>   <DIVIDE>

### 三、 系统分析和设计

#### (一) 实验环境

1. OS: Ubuntu Kylin 15.04
2. IDE: JetBrains Pycharm 4.5.4
3. Programming Language : Python2.7
4. GUI Library : Qt 5.2

#### (二) 系统架构设计

- 逻辑部分：
  1. tokens.py：词法单元，预定义的 Token 类
  2. lexer.py：词法分析器
  3. nodes.py：语法树节点， 包含语法、语义、中间代码生成等子模块
  4. stable.py：语义分析器，符号表等
  5. parser.py：语法分析，语义分析，中间代码生成等总控程序
  6. inter.py：解释器，解释执行中间代码

- 界面部分

界面先在 QtCreator 中设计，生成 **window.ui** 界面文件以及 **cinter.qrc** 资源文件

1. window.py：利用 **pyuic5** 工具，由 window.ui 生成的界面文件
2. cinter\_rc.py: 利用 **pyrcc5** 工具，由 cinter.qrc 生成的资源文件
3. editor.py：继承 QPlainTextEditor, 实现行号显示，高亮当前行等功能
4. highlighter.py：实现代码高亮
5. window.py：程序主窗口

### (三) 系统功能设计

- 逻辑部分

1. 在 `parser.py` 中调用 `lexer.py` 进行词法分析，将分析得到的词法单元用于 `parser.py` 中的语法分析。
2. `nodes.py` 中定义了语法单元中所有非终结符对应的语法树节点，在语法分析中会不断地构造这些节点，最终生成完整的抽象语法树。
3. 上述语法树节点中包含有语义分析，中间代码生成的功能，只需要分两次遍历语法树即可分别完成语义分析以及中间代码生成。
4. 在语义分析过程中会使用到 `statble.py` 中的符号表模块生成符号表，以及利用其中的类型检查等模块进行语义分析。
5. 遍历抽象语法树生成中间代码后，调用 `inter.py` 中的中间代码解析器即可完成其解释执行。

- 界面部分

1. 界面分为菜单栏、工具栏、功能区。
2. 功能区分为三个部分，上左为文件浏览器、词法、语法分析结果，使用 Tab 页的方式进行切换；上右为代码编辑器，支持打开多文件，以 Tab 页的方式进行切换。下面是控制台，支持关闭，拉伸等功能。
3. 以上各个部分都支持最大化，隐藏、重现等操作。
4. 编辑器支持文件新建、打开、保存、另存为以及常用复制、粘贴，剪切等功能。
5. 程序的输入与输出都重定向到底部的控制台，不需要调用系统输入输出界面模块。

## 四、 功能模块详解

### (一) 词法分析

#### 1. 源代码读入

待处理的源代码可以从键盘读入也可以从文件读入。对于文件读入方式，`parser`只需要把打开后的文件对象传递给`lexer`即可。而对于键盘读入，为了将两种方式进行统一，采取的方式是先将键入的源代码全部读取完毕，然后将读入的字符串转换为`file-like object`。`file-like object`的读取方式与真实的文件对象一致，从而完成了两种输入方式的统一。

#### 2. 词法单元识别

首先，需要根据 CMM 词法特性构造出其对应的`DFA`，然后再根据`DFA`构造其状态转换表，再根据状态转换表构建分析程序。`DFA`中，每个结束状态都表示成功识别一个词法单元。当成功识别一个词法单元之后，重新回到 DFA 中的开始状态，再次进行识别，直到`EOF`为止。

理论上，需要维护一个状态转换表，其结构为一维数组。数组角标对应于 DFA 中的状态编号，其值为一个`dict`，`dict`中存储了状态转换键值对。其中，`key`对应于 DFA 中的转移字符，`value`对应于转换后的状态。终止状态对应于其识别到的`Token`。然而，由于构建的`DFA`比较简单，`lexer`处理程序中并没有机械性地根据状态转换表进行识别，而是根据`DFA`直接采用传统的`if-else`结构进行词法分析。

#### 3. 字符读取

`lexer`处理程序会将源代码中的字符逐个读取，并逐个识别。当读到一个“脏字符”时，标志者当前的`TOKEN`识别结束，重新回到`DFA`的开始状态。而重新开始识别时，需要再次使用到之前读取的“脏字符”。因此，需要维护一个输入缓冲区。

缓冲区其实是一个堆栈，FILO。当读到“脏字符”时，我们将其放入缓冲区，并结束当前的识别，返回识别到的`Token`。接着，从`DFA`的开始状态重新开始识别。再次读取字符时，需要先判断缓冲区是否为空，若缓冲区还有字符，则将缓冲区中的字符依次弹栈，直到清空为止。待缓冲区为空后，再从输入源读取字符。

#### 4. 错误处理

当遇到非法字符时，需要进行错误处理。本程序中，采用打印错误信息并立即终止程序的方式。打印的错误信息包含非法字符所在行、列，以及错误行的所有内容。比如，遇到`0..04545`时，错误信息为

```
Invalid token at row 5, column 9:  
  
int a=0..4545 ;  
  
      ^
```

因此，需要使用几个变量记录当前的信息。`line`记录当前行号，`read`记录当前行已经读取过的所有字符。在读到非法字符时，`line`确定非法字符所在行，`read`的大小确定非法字符所在列。继续读完当前行未读完的部分，并将其存入`read`中，即可打印当前行。

当读取一个字符时，不论是从缓冲区还是从源文件，都应该将其加入`read`；当将“脏字符”放入缓冲区时，同时也应该去除`read`中读到的脏字符；当遇到换行符时，`line`递增，并将`read`清空。

若读到换行符并进行递增和清空等换行操作后，发现换行符是“脏字符”，则需要把换行符放入缓冲区并回到上一行。要是非法字符刚好出现在上一行的末尾，如`ab2c\_`中的`\_`，则需要打印上一行的所有字符并报错。但是上一行记录的所有字符已经在进行换行操作时被清空，因此我们需要再使用一个变量`pre\_read`来记录换行时上一行的所有字符。



## (二) 语法分析

### 1. 分析方法

语法分析其实就是按照事先定义好的语法，对词法分析器返回的 Token 与语法元素进行匹配，若是发现不匹配则报错。本程序采用**递归下降法**进行分析。

若用产生式定义语法，则每个非终结符对应一个处理函数，终结符对应从词法分析器获取的 Token。

### 2. 分析过程

- 1) 从词法分析器获取一个 Token，根据其词素判断其属于哪一种语句。
- 2) 进入该非终结符的处理程序。根据**语法定义**，若语法定义中接下来是一个非终结符，则进入该非终结符的处理函数；若是一个终结符，则判断其是否与获取的 Token 相匹配。如此往复进行匹配，处理完一条语句则返回过程 1
- 3) 匹配过程中，若不相符，则抛出异常，终止分析程序。
- 4) 处理过程中生成语法分析树，每个非终结符的处理函数都会返回对应的语法树节点。

### 3. 主要数据结构

#### 1) 向前看缓冲区

在匹配的过程中会遇到可选的表达式，比如 if-else 结构中的 else 可以省略。当匹配 else 失败时，需要将当前获取的 Token 放入缓冲区，以便下次判断时使用。

#### 2) 当前行记录

分析过程中需要记录当前所在源代码的行数，以便出错时进行报错。

#### 3) 语法树节点

每个非终结符的处理函数在结束时会返回其对应的节点对象。每个节点的属性中包含其父节点以及其子节点列表。

## 4. 主要算法

### 1) 期待算法

定义一个 `expect(Token)` 函数，传入期待遇到的 Token。在函数中，会获取接下来的一个 Token，并与传入的参数进行匹配。若不匹配，则抛出异常，停止分析程序。

### 2) 匹配算法

定义一个 `match(Token)` 函数，传入要匹配的 Token。在函数中，会获取接下来的一个 Token，并与传入的参数进行比较。若相同，则返回 `True`，否则返回 `False`

### 3) 缓冲算法

将匹配过程中获取到的不匹配但将来还会使用的 Token 放入缓冲区。下次从词法分析器获取 Token 时，先判断缓冲区内是否为空，若为空则从词法分析器获取，若不为空则从缓冲区获取。

### 4) 语法树生成

每个非终结符都对应一个处理函数，每个处理函数结束时都会返回一个语法数节点。语法树节点与其父节点关联，同时，也保存有其所有子节点的引用。按照语法定义的顺序，依次将子节点添加到父节点上。节点基类定义如下

```
class Node(object):
    """
    Basic node which is also used in QAbstractItemModel.
    """
    def __init__(self, cate, parent=None):
        self.cate = cate # for print
        self.parent = parent
        self.childItems = []
```

各个节点以递归的方式生成。比如在处理 `expression` 时，它首先要进入 `term` 的处理函数，得到其返回的 `TermNode` 之后才能生成 `ExprNode` 节点。

处理 expression 的函数如下：

```
def _parse_expr(self):
    """
    expression ::= term (addOp term)*
    :return:
    """
    term = self._parse_term()
    l = []
    while True:
        add = self._match_op_add()
        if add:
            t = self._parse_term()
        else:
            self._unget()
            break
        l.append((add, t))
    return ExprNode(term, l)
```

可以看到在生成 ExprNode 时需要传入 \_parse\_term 返回的 TermNode 节点。

ExprNode 的构造函数如下：

```
class ExprNode(Node):
    """
    expression ::= term (addOp term)*
    """

    def __init__(self, term, addOp_term_list=None):
        super(ExprNode, self).__init__('Expression')
        self.append(term)
        if addOp_term_list:
            for pair in addOp_term_list:
                self.append(pair[0])
                self.append(pair[1])
```

程序**并不是**先定义一个根节点，然后在各个处理函数中生成节点，并将生成的节点附加到根节点上。这种添加的方式会使得代码结构比较混乱，我们可以按上述方法把语法分析与抽象语法树的生成抽离开来，把语法树的生成、连接放到各个定义好的节点类的构造函数中。处理函数只负责生成其对应的节点，并将节点返回给上层的处理函数进行处理。

这样，每个非终结符的处理函数都可以生成一颗以其自己为根的语法树。在进行语法分析时只需要调用最顶层的非终结符`program`的处理函数，则可以得到一颗以其为根的

语法树。

## 5) 语法树输出

由于每个节点都与其父节点以及所有子节点关联，可以对其根节点采用**深度优先搜索**

算法对所有节点进行遍历。并将其词素打印输出。示例如下：

```
Console [x]
/home/finalize/Workspace/pycharm/cinter/test/2_parser/array.t
|----> ExtStmts
|   |----> DeclareStmt
|   |   |----> INT : "int"
|   |   |----> Array
|   |   |   |----> INT_LITERAL : "5"
|   |   |   |----> ID : "a"
|   |   |   |----> ArrayInitNode
|   |   |       |----> INT_LITERAL : "1"
|   |   |       |----> INT_LITERAL : "2"
|   |   |       |----> INT_LITERAL : "3"
|   |----> DeclareStmt
|   |   |----> INT : "int"
|   |   |----> ID : "i"
|   |----> DeclareStmt
|   |   |----> INT : "int"
|   |   |----> ID : "j"
|   |----> FuncDefStmt
|   |   |----> ReturnType: VOID
|   |   |----> ID : "main"
|   |   |----> FuncDefParams: VOID
|   |   |----> InnerStmts
|   |       |----> AssignStmt
|   |       |   |----> ID : "j"
|   |       |   |----> Expression
|   |       |       |----> Term
|   |       |       |----> Factor
```

## (三)语义分析

语义分析在生成抽象语法树之后进行。从抽象语法树的根节点开始，遍历整个抽象语法树。遍历的同时构建符号表，在需要进行语义分析之处根据符号表进行分析。

### 1. 符号表结构

程序采用多层符号表，属于不同作用域的符号分别在不同的符号表中。

#### 1) 符号的结构

符号表中的每个元素都是一个 Symbol 对象。其中包含名称 ( name )，类型

( stype )，词素 ( lexeme )，以及所在的符号表 ( table )。定义如下：

```

class Symbol(object):
    def __init__(self, name, stype):
        assert isinstance(name, basestring)
        assert isinstance(stype, SType)

        self.name = name
        self.stype = stype
        self.lexeme = None
        self.table = None

```

## 2) 符号的类型

每个符号 ( Symbol ) 有其对应的类型 ( 基类为 SType )。有以下四种类型：

- SType：基本数据类型，如 int real
- STypeArray：数组类型
- STypeFunc：函数，其属性包含函数的参数列表对应的类型
- SUnknown：语义分析过程中临时采用的未知类型

## 3) 符号表的结构

类似于树形结构，符号表通过持有上层符号表的引用来实现各层次符号表的联系。同属一层的符号表并不能相互访问，只有同属**同一个**符号表的符号才能相互访问。另外，可以访问上层符号表中的符号。但符号的定义有先后顺序，在访问上层符号表时只能访问已经定义过的符号。因此需要记录生成子符号表时的位置。

符号表包含以下属性：

- parent：上一层符号表
- children：在本符号表范围内定义的子（下一层）符号表
- symbols：符号表中的符号
- tsindex：符号 smb 在上一层符号表中的标号。其中 smb 是上一层符号表中的符号，定义符号 smb 之后，紧接着生成了本符号表。即为本符号表在上

一层符号表中的位置。

- children\_tsindex : 所有子符号表的位置

符号表 ( STable ) 的构造函数如下：

```
class STable(object):
    def __init__(self):
        self.parent = None # parent table
        self.children = [] # children tables
        self.symbols = [] # symbols in the table

        # The Symbol index in parent
        # after which the table was appended
        self.tsindex = -1
        self.children_tsindex = [] # tsindex of children
```

#### 4) 符号表示例

下图中，符号 “--->o” 表示其为符号表的**第一个符号**。可以看出，符号 **e** 与符号 **i** 各自所在的符号表属于同一层，但是处于不同的符号表，因此其相互之间不能访问。符号 **w** 能访问其上层符号表中的符号 **a**，以及符号 **arr**。符号 **cc** 与符号 **a** 属于同一个符号表。

<pre>int[3] arr; void main(){     int a=0;     if (a&gt;0){         while(a&gt;0){             int w;         }     }else{         real e;     }     if (a==0){         int i;     }     int cc;     return ; }</pre>	<p>符号表打印如下：</p> <pre> ---&gt;o(read:int)  ---&gt;(write:void)  ---&gt;(arr:int)  ---&gt;(main:void)  -----&gt;o(a:int)  -----&gt;o(w:int)  -----&gt;o(e:real)  -----&gt;o(i:int)  -----&gt;(cc:int) Process finished successfully</pre>
---	---

( 表中 read write 是内建函数，放在符号表的最顶层 )

## 2. 符号表构建

通过遍历抽象语法树进行符号表构建。每个语法树节点都实现了 **gen\_stype** 与 **gen\_stable** 方法。gen\_stype 用于生成该节点的类型 ( SType ), gen\_stable 用于生成新的子符号表或者进行语义分析。二者在所有节点的基类 Node 中声明, 子类继承并重载即可。定义如下:

```
def gen_stable(self, stable):  
    """  
    Append a new variable to symbol table or add a new table  
    For generate symbol table and semantics analysing  
    :param stable:  
    :return:  
    """  
    assert isinstance(stable, STable)  
    return None  
  
def gen_stype(self):  
    """  
    Generate the symbol type in symbol table.  
    For semantics analysing.  
    :return: **must return a list**  
    """  
    return []
```

遍历语法树的过程中, 每个节点都会调用 gen\_stable 方法, 并传入其**所在的符号表** ( 即参数 stable )。若在此节点所表示的语法规则中会新建一个符号表, 则将新建的符号表与父符号表进行关联, 然后返回新建的符号表。若语法规则中有需要进行语义分析的部分, 则调用 stable 相应的语义分析程序进行分析, 在下一节详述。也可以直接返回 None 不进行任何操作 ( 基类 Node 中定义, 子类不重载表示使用此方法 )。最常见的是将一个符号放入符号表

### ■ 示例 1: 函数调用: 只进行语义检查

```
def gen_stable(self, stable):  
    stable.invoke_func(self.name,  
        self.params.gen_stype() if self.params else [])
```

■ **示例 2：innerStmtsNode**：新建符号表

```
def gen_stable(self, stable):
    table = STable()
    stable.table_append(table)

    # add function param list to stable
    if isinstance(self.parent, FuncDefStmtNode):
        self.parent.def_param(table)
    return table
```

■ **示例 3：函数声明**：将函数对应的符号加入符号表

```
def gen_stable(self, stable):
    stable.symbol_append(self.funcId.gen_symbol())
```

### 3. 语义分析

上面说到遍历语法树时（gen\_stable）会将该节点所在的符号表实例（stable）作为参数传入。符号表对象（STable）实现了多种语义分析函数用来分析。涵盖以下几种错误类型：

- 函数、变量等未定义
- 函数、变量等重定义
- 类型不匹配（不支持类型转换）
- 函数参数个数不匹配
- 函数参数类型不匹配
- 函数返回值与返回值类型不匹配
- 数组角标缺失、非整数
- 数组初始化时溢出
- 左值右值错误使用等
- 主函数返回值为 VOID



其中，核心部分是标识符的**类型确定与查询**。下面会分别介绍其解决方案。

## 1) 类型确定与匹配

抽象语法树的每个节点类都会实现或继承 `gen_stype` 方法，此函数会生成该节点对应的 `SType`。对于数组类型 (`SArrayType`)，其包含数组大小的成员变量，对于函数类型 (`STypeFunc`)，其属性中包含返回值类型、参数个数、参数类型等信息。

在遍历过程中，遇到需要进行语义分析的节点，则调用其子节点的 `gen_stype` 方法，将所有需要分析的 `SType` 交给符号表实例进行检测。符号表 (`STable`) 实例实现了以下几个分析函数：

```
m invoke_assign(self, name, stype_list, is_arr=False)
m invoke_compare(self, stype_list1, stype_list2)
m invoke_func(self, name, param_stype_lists)
m invoke_return(self, stype_list=None)
```

他们接收涉及到的标识符或者是符号类型，然后进行类型匹配等语义分析。若遇到语义错误则会抛出异常。

## 2) 查询符号表

进行标识符的类型确定时，往往需要查询符号表。查询符号表的原则是：

- 先查询当前符号表
- 若当前符号表不存在该符号，则查询上一层符号表。
- 从最新定义的符号开始查询，也就是从后往前遍历 `STable.symbols` 列表
- 从符号表在上级符号表所在位置开始，往前遍历。不能遍历到在其之后定义的变量，也就是当前未定义的变量。

以上要求可以递归实现。在下面的程序中，先遍历 `self.symbols` 也就是当前符号表。然后再递归地调用上一层符号表的 `_symbol_find` 函数，直到找到或遍历完所有

符号为止。具体实现如下：

```
def _symbol_find(self, name, ends=None):
    """
    find the symbol defined before with name of 'name'

    :param ends: searching in [0,ends] in the symbol list of current table.
                  Used when searching in parent table whose value is 'tsindex'
    :return: the symbol or None
    """
    if not ends:
        ends = len(self.symbols) - 1

    for i in range(ends, -1, -1): # check self
        smb = self.symbol_at(i)
        if smb.name == name:
            return smb
    else: # check parent tables
        if self.parent:
            return self.parent._symbol_find(name, self.tsindex)
        else: # recursion ends at root table whose parent is None
            return None
```

**ends** 表示子表在父表中的插入位置

### 3) 遍历代码实现

```
# the node and the direct symbol table which it is in
stack = [(self.rootNode, self.stable)]
while len(stack) > 0:
    node, stable = stack.pop()
    try:
        table = node.gen_stable(stable)
    except SemanticsError, e:
        self.stderr.write('%s %s\n' % (str(e), node.gen_location()))
        return None
    else:
        children = list(node.childItems)
        children.reverse()
        children = [(child, table or stable) for child in children]
        stack += children
```

其中 SemanticsError 是自定义语义错误的基类

## (四) 中间代码生成

抽象语法树的每个节点类都继承或重载了生成中间代码的方法，只需要遍历一遍抽象

语法树即可以生成中间代码。下面是基类 Node 中定义的代码生成方法：

```
def gen_code(self):
    """
    Generate Intermediate Code
    :return: **must return a list**
    """
    return []
```

因为每个语法树节点生成的中间代码不只一行，为了类型统一，要求 **gen\_code** 函数必须返回一个列表。下面介绍中间代码的结构与类型。

1. 中间代码的结构

在 **inter.py** 中，定义了 **Code** 类。一个 Code 实例表示一行中间代码。每个语法树节点都会生成一个 Code 列表，包含其对应的中间代码。当调用根节点的 **gen\_code** 方法时，会自底而上地生成并返回所有子节点生成的代码。在父节点中，只需要按照语法逻辑，将子节点生成的代码串联起来即可。

将 Code 对象输出为字符串，形式为 ( op,arg1,arg2,tar )

- op：指令，如 算数运算、跳转、等
- arg1 arg2：操作数 1、2
- tar：目标对象，将运算结果存到其中或者对该对象进行操作

2. 四元式类型

一共有以下几种四元式类型

四元式含义	四元式	备注
声明	( = , _i   _r , , var )	_i 指 int, _r 指 real
赋值	( = , var1, , var2 )	
函数定义	( f= entrance, , ,func )	func 为函数名 entrance 为函数入口地址
数组声明	( = , _i[]   _r[] , , var )	_i 指 int, _r 指 real

数组赋值	( []=, index , value ,arr )	arr 为数组 , index 为角标
数组索引	( =[, arr, index, var )	arr 为数组 , index 为角标
四则运算	( +   -   *   / , arg1,arg2, var )	
跳转	( j , , , address )	address 为目标地址
条件跳转	( j>   j<   j==   j<> , , , address )	address 为目标地址
参数传入	( p=, value, , param )	param 为参数名称
参数获取	( =p, param, , var )	param 为参数名称
函数调用	( c , , , func )	Func 为函数名称
函数返回	( r , , , )	

### 3. 特殊变量

- 返回地址：\_ra
- 返回值：\_rv
- 函数参数：\_pi 其中 i 是参数的序号，从 0 开始

### 4. 拉链反填实现

当遇到 If, while, 以及函数等需要跳转但暂时不知道跳转地址的情况时, 需要使用拉链反填。其实也就是先不填所缺的地址, 等到可以知道地址之后再返回去修改。在程序中很容易实现。拿 while 举例:

当处理 condition 时并不知道要跳转到哪里, 只能先将代码返回给 WhileStmt

```
def gen_code(self):
    codes = []
    arg1 = self.childAt(0).gen_code()
    op = self.childAt(1).gen_code()
    arg2 = self.childAt(2).gen_code()

    codes += arg1
    codes += arg2
    # unknown jump target, so set -1
    link = Code(op=op, arg1=arg1[len(arg1) - 1].tar,
               arg2=arg2[len(arg2) - 1].tar, tar=-1)
    codes.append(link)
    return codes
```

然后再到 WhileStmt 中进行处理：计算出循环体的代码行数，结合 condition 返回的

最后一行的代码的行号，就可以确定跳转地址：

```
def gen_code(self):
    cond = self.childAt(0).gen_code()
    stmts = self.childAt(1).gen_code()
    # backfill the jump address
    cond[len(cond) - 1].tar = stmts[len(stmts) - 1].line + 2
    # go back to cond
    return cond + stmts + [Code(op='j', tar=cond[0].line)]
```

## 5. 举例分析

### 1) 函数声明与返回

<pre>int foo(int a,int b){     return a+b; }  void main(){     foo(1,2);     return; }</pre>	<pre>0: ( f= , 2 , , foo ) 1: ( j , , , 11 ) 2: ( = , _i , , a ) 3: ( = , _i , , b ) 4: ( =p , _p0 , , a ) 5: ( =p , _p1 , , b ) 6: ( = , a , , _t6 ) 7: ( = , b , , _t7 ) 8: ( + , _t6 , _t7 , _t8 ) 9: ( = , _t8 , , _rv ) 10: ( r , , , ) 11: ( f= , 13 , , main )</pre>
--	---

上图截取了该程序生成的中间代码中函数 **foo** 的代码。

## ■ 函数声明：

第 0 行，声明了函数 `foo`，其入口地址为 2。第 1 行，条转指令，跳转到 11 行，也就是 `main` 函数的定义部分。因为中间代码是**顺序执行的**，遇到函数声明时控制流并不会进入函数体，而是会跳过函数定义，等调用该函数时再进入函数体。因此需要在函数入口（第 2 行）前跳转出函数体（2-10 行为函数体）

## ■ 函数返回

第 9,10 两行是函数返回语句。第 9 行设计了一个特殊变量 `_rv`，用于保存函数返回值。调用者在函数调用之后再取出其值即可获取返回值。第 10 行只有一个返回命令，却没有返回地址。因为在函数调用时就已经将返回地址存在函数调用帧中，此处会在第（五）节详解。

## 2) 函数调用与参数传递

### ■ 参数获取

还是上面的图，2-5 行是参数声明与获取。函数的参数应该声明在函数的作用域中，故有 2,3 两行。在函数调用时，并没有把参数放入函数调用栈中，而是将参数传递到临时变量 `_pi` 中，此处 `i` 是参数的序号，从 0 开始。故在此函数中，`a` 需要接收 `_p0`，`b` 需要接收 `_p1`。

### ■ 参数传递

<pre>int foo(int a,int b){     return a+b; }  void main(){     foo(1,2);     return; }</pre>	<pre>11: ( f= , 13 , , main ) 12: ( j , , , 22 ) 13: ( = , 1 , , _t13 ) 14: ( p= , _t13 , , _p0 ) 15: ( = , 2 , , _t15 ) 16: ( p= , _t15 , , _p1 ) 17: ( = , 19 , , _ra ) 18: ( c , , , foo ) 19: ( = , _rv , , _t19 )</pre>
--	--

同样的程序，第 13-18 行是调用 `foo(1,2)` 的中间代码。依次将 1,2 赋值给 `_p0, _p1`，作为参数传递。上面已经介绍到，在函数体中会将参数的值取出。第 17 行，将返回地址赋给了一个特殊变量 `_ra`，用于记录执行完函数后的返回地址。第 18 行调用 `foo`，会在符号表中查找其入口地址。

## (五) 中间代码解释执行

遍历抽象语法树之后会生成一个 Code 示例的列表，每个 Code 实例对应一条四元式。只需要依次遍历 Code 列表，按照四元式中的指令进行运算、跳转，直至遍历完成。跳转指令对应的地址也就是 Code 实例在列表中的索引，最终会跳转到一个不存在的索引，停止程序的执行（即为列表长度）。

### 1. 主要数据结构

#### 1) 符号表

对于全局变量，以及函数帧中声明的变量，需要分别为其维护一个符号表。符号的结构如下：

```
class Symbol(object):
    """
    Symbol
    """
    type_int = 0
    type_real = 1
    type_func = 2
    type_read = 3

    def __init__(self, name, _type, value=None, size=-1):
        self.name = name
        self.type = _type
        self.size = size
```

由于已经完成语义分析，可以简单地按照 `_type` 的值来区分不同类型的符号。

`size` 不为 -1 则表示该符号为数组。

## 2) 函数帧

函数调用时需要为每个函数维护一个函数帧，函数声明的变量都需要放到该函数帧对应的符号表中，而不能放入全局符号表。在初始化函数帧时，需要传入函数的返回地址。

```
class Frame(object):  
    """  
    Function frame  
    """  
  
    def __init__(self, raddress):  
        self.symbols = []  
        self.raddress = raddress # return address
```

## 3) 函数调用栈

对于任何一次函数调用，都需要将其函数帧放入调用栈中，当函数返回后将其弹出。

# 2. 主要算法

## 1) 符号管理

### ■ 符号添加

为全局变量维护一个符号表，并为每个函数帧维护各自的符号表。在同一个符号表中，没有层次之分，只是按定义顺序依次将符号添加到符号表中。若出现重定义的现象，比如在两个并列的 if 语句中分别定义一个同名的变量，只需要在第二次定义时将其值清零即可。

### ■ 符号搜索

首先搜索当前函数帧的符号表，若未找到，则在全局符号表中寻找。



## 2) 执行方式

由于每个符号都有其对应的 value，数组对应的 value 是一个列表，只需要不断地按照四元式中的指令对各个符号的值进行更新即可。等到要使用时再取出。执行程序使用一个循环遍历所有的四元式，每遍历一次 line 递增一次。循环体内只需要根据不同的指令做出相应的计算。若遇到跳转指令，更改 line 的值为要跳转到的地址即可。执行到主函数的返回语句时结束执行程序。

```
while line < len(self.codes) - 1:
    code = self.codes[line]
    op = code.op
    arg1 = code.arg1
    arg2 = code.arg2
    tar = code.tar

    if op == '=':
```

## 3) 函数调用

函数调用时，新建一个函数帧，以及其对应的符号表，并把函数帧加入到函数栈中。

对于参数传递，只需要将参数加入到全局符号表中，在函数体中获取全局的参数即可。

## 4) 输入输出

提供 write 以及 read 两个内建函数。当遇到 write 函数时，不进行函数栈的操作，只需要将传入的参数取出，然后输出，并将 line 设置为返回地址即可。对于 read，与 write 大体相同，只需要从前端界面的控制台获取输入即可。

## 5) 错误检测

执行的过程中可能会引发以下几种错误：

- 除零（ZeroDivisionError）
- 数组角标越界（IndexError）
- read 读入的数据类型不匹配（ValueError）

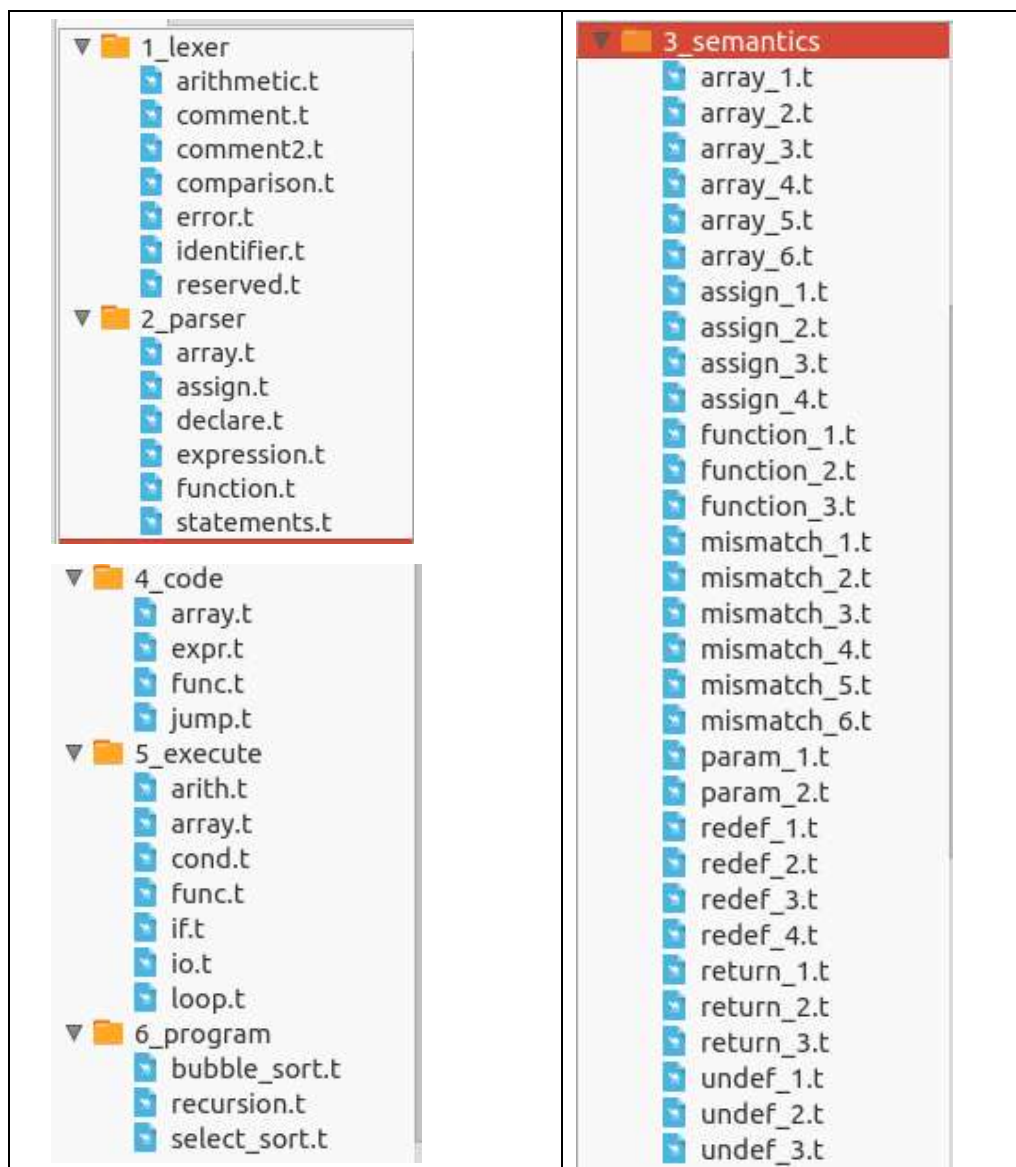
因此，需要增加一个异常处理机制，当检测到上述异常时，跳出循环，停止中间代码的遍历，结束程序的运行。并在控制台上打印错误。

## (六) 界面设计

UI 界面与程序的执行在不同的线程中，而程序输出是在程序执行线程内，因此需要实现在非 UI 线程更新 UI。利用 Qt 的信号和槽机制，在向控制台输出时发送一个信号，在 UI 线程中接收信号，在槽中更新 UI。

## 五、 测试用例和测试结果

共有如下测试用例。下面只展示部分测试用例，全部测试用例可在项目源码中获取。



## (一)词法分析

### 1. 注释

The screenshot shows a lexical analyzer interface. On the left, a tree view shows the tokenization of the code. The main window displays the source code with line numbers 1 through 12. The code includes a multi-line comment, a single-line comment, and a nested comment. The tokenized output is shown on the right, with tokens like <RESERVE: 'i...>, <ID: 'i'>, <ASSIGN: '='>, <INT\_LITERA...>, <SEMICOLO...>, <ID: 'ccc'>, <TIMES: '\*'>, and <DIVIDE: '/'>.

```
1  /**
2   * This is a test for multiple line
3   * comments:
4   */
5  int i/*~ /*Ha*ha/* ~*/=0;
6
7  /*
8   Below is a nest comment, which is
9   invalid.
10  But at lexical analysis state, it's
11  valid,
12  which can be recognized as "<ID: 'ccc'
13  <TIMES: '*'> <DIVIDE: '/'>"
14  */
15
16  /*aaa/*bbb*/ccc*/
```

### 2. 算数运算

The screenshot shows a lexical analyzer interface. On the left, a tree view shows the tokenization of the code. The main window displays the source code with line numbers 1 through 6. The code includes a multi-line comment, a single-line comment, and a nested comment. The tokenized output is shown on the right, with tokens like <ID: 'a'>, <PLUS: '+'>, <ID: 'b'>, <SEMICOLON: ';'>, <ID: 'a'>, <MINUS: '-'>, <ID: 'b'>, <SEMICOLON: ';'>, <ID: 'a'>, <TIMES: '\*'>, <ID: 'b'>, <SEMICOLON: ';'>, <ID: 'a'>, <DIVIDE: '/'>, <ID: 'b'>, and <SEMICOLON: ';'>.

```
1  // This is a test for arithmetic
2  a+b; a- b; a*b; a / b;
3
4  int b=0.00134446
5  int a=0.4545 // invalid
6
```

### 3. 报错



```
1 //This is a test for identifier
2 a2b_c // valid
3 a2b_ // invalid
```

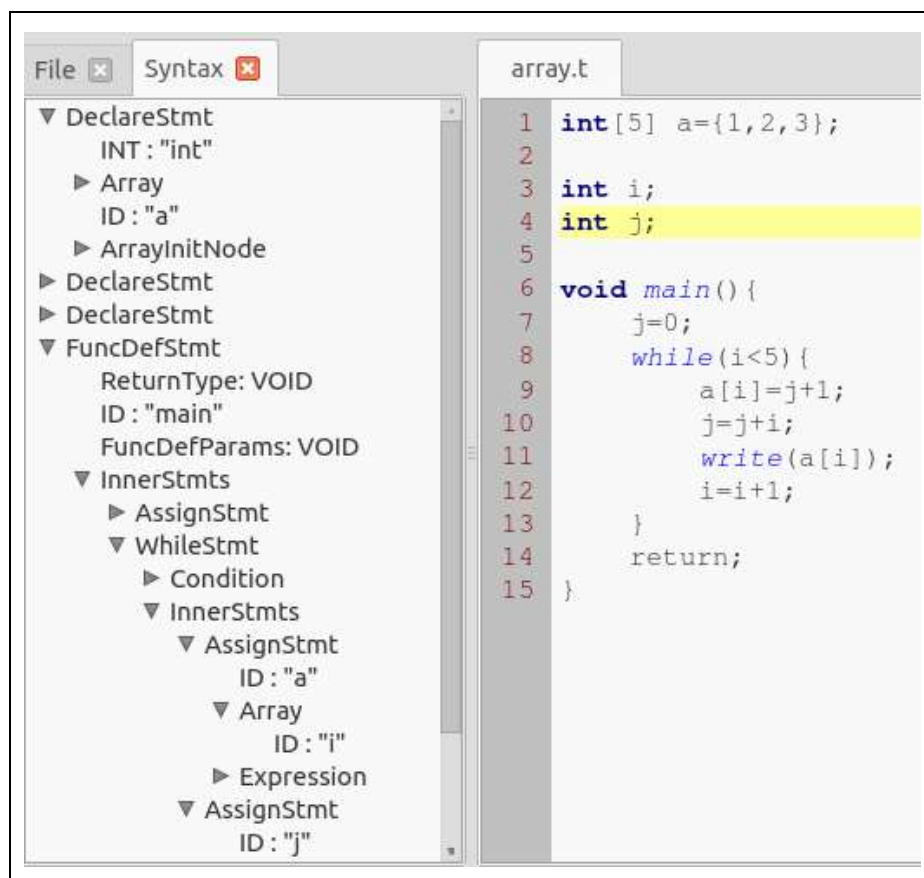
Console ✖

```
/home/finalize/Workspace/pycharm/cinter/test/
1_lexer/identifier.t

Invalid token at row 3, column 4:
a2b_ // invalid
  ^
Process finished unsuccessfully
```

## (二)语法分析

### 1. 数组



File ✖ Syntax ✖ array.t

**Syntax Tree:**

- ▼ DeclareStmt
  - INT: "int"
  - ▶ Array
    - ID: "a"
  - ▶ ArrayInitNode
- ▶ DeclareStmt
- ▶ DeclareStmt
- ▼ FuncDefStmt
  - ReturnType: VOID
  - ID: "main"
  - FuncDefParams: VOID
  - ▼ InnerStmts
    - ▶ AssignStmt
    - ▼ WhileStmt
      - ▶ Condition
      - ▼ InnerStmts
        - ▼ AssignStmt
          - ID: "a"
          - ▼ Array
            - ID: "i"
          - ▶ Expression
        - ▼ AssignStmt
          - ID: "j"

**array.t**

```
1 int[5] a={1,2,3};
2
3 int i;
4 int j;
5
6 void main(){
7     j=0;
8     while(i<5){
9         a[i]=j+1;
10        j=j+i;
11        write(a[i]);
12        i=i+1;
13    }
14    return;
15 }
```

## 2. 赋值

The screenshot shows a code editor with a file named 'assign.t'. The code is as follows:

```
1 // This is a test for assign
2
3 int a;
4 real[3] b,d;
5
6 int foo(){
7     return 1;
8 }
9 void main(){
10     a=foo();
11
12     b[0]=1.5;
13     b[1]=a;
14     b[2]=4.0;
15
16     d=(b[0])+(b[3]*a)/5;
17     return;
18 }
```

The AST on the left shows the following structure:

- ▼ DeclareStmt
  - INT : "int"
  - ID : "a"
- ▼ DeclareStmt
  - REAL : "real"
  - Array
    - ID : "b"
    - ID : "d"
- ▼ FuncDefStmt
  - ReturnType
  - ID : "foo"
  - FuncDefParams: VOID
  - InnerStmts
- ▼ FuncDefStmt
  - ReturnType: VOID
  - ID : "main"
  - FuncDefParams: VOID
  - ▼ InnerStmts
    - AssignStmt
    - AssignStmt
    - AssignStmt
    - AssignStmt
    - AssignStmt

## 3. 表达式

The screenshot shows a code editor with a file named 'expression.t'. The code is as follows:

```
1 //This is a test for arithmetic
2
3 int a;
4 real b;
5
6 real foo(){
7     return 0.0;
8 }
9
10 void main(){
11     b=3.14159265354/foo()
12     +2.718281828459045;
13     a=((a+2)*6-(b/2048+3))/9*(9+0-b);
14     return;
15 }
```

The AST on the left shows the following structure:

- FuncDefStmt
- ▼ FuncDefStmt
  - ReturnType: VOID
  - ID : "main"
  - FuncDefParams: VOID
  - ▼ InnerStmts
    - ▼ AssignStmt
      - ID : "b"
      - ▼ Expression
        - ▼ Term
          - ▼ Factor
            - REAL\_LITERAL : "3.14159265354"
          - ▼ Multiply
            - DIVIDE : "/"
            - ▼ Factor
              - ▼ FuncCallExpr
                - ID : "foo"
          - ▼ Add
            - PLUS : "+"
          - ▼ Term
            - ▼ Factor
              - REAL\_LITERAL : "2.71828182846"
      - ▼ AssignStmt
        - ID : "a"
        - ▼ Expression
          - ▼ Term
            - Factor
            - Multiply
            - Factor
            - Multiply
            - Factor
  - ReturnStmt

## 4. 语句

statements.t	
<pre> 1 // this is a test for all the statements 2 3 real b; 4 5 int func(int a){ 6     return a*a; 7 } 8 9 int a; 10 int[3] arr; 11 12 void main(){ 13     a=read(); 14 15     while(1==(a+9*8-arr[0])){ // `while` test 16         write(9); 17         if (a&gt;9){ // `if-else` test 18             write(func(a)-func(a)/func(a)); 19         }else{ 20             if(a&lt;15){ // `if` test 21                 write(a+9/arr[1]); 22             } 23         } 24     } 25     return; 26 } </pre>	<ul style="list-style-type: none"> <li>▼ DeclareStmt             <ul style="list-style-type: none"> <li>REAL : "real"</li> <li>ID : "b"</li> </ul> </li> <li>▼ FuncDefStmt             <ul style="list-style-type: none"> <li>▼ ReturnType                     <ul style="list-style-type: none"> <li>INT : "int"</li> </ul> </li> <li>ID : "func"</li> <li>► FuncDefParams</li> <li>► InnerStmts</li> </ul> </li> <li>► DeclareStmt</li> <li>► DeclareStmt</li> <li>▼ FuncDefStmt             <ul style="list-style-type: none"> <li>ReturnType: VOID</li> <li>ID : "main"</li> <li>FuncDefParams: VOID</li> <li>▼ InnerStmts                     <ul style="list-style-type: none"> <li>▼ AssignStmt                             <ul style="list-style-type: none"> <li>ID : "a"</li> </ul> </li> <li>▼ Expression                             <ul style="list-style-type: none"> <li>▼ Term                                     <ul style="list-style-type: none"> <li>▼ Factor   <ul style="list-style-type: none"> <li>▼ FuncCallExpr   <ul style="list-style-type: none"> <li>ID : "read"</li> </ul> </li> </ul> </li> </ul> </li> </ul> </li> </ul> </li> <li>▼ WhileStmt                     <ul style="list-style-type: none"> <li>▼ Condition                             <ul style="list-style-type: none"> <li>▼ Expression                                     <ul style="list-style-type: none"> <li>► Term</li> </ul> </li> <li>▼ Compare                                     <ul style="list-style-type: none"> <li>EQUAL : "=="</li> </ul> </li> <li>▼ Expression                                     <ul style="list-style-type: none"> <li>► Term</li> </ul> </li> </ul> </li> </ul> </li> </ul> </li> <li>▼ InnerStmts</li> </ul>

## 5. 函数


<pre> // function define test void void_test(void){     return; }  int void_test2(){     return 1; }  int normal_test(int a){     return b+a; }  void multi_args_test(int a, real b){      return; }  int main(){     int a=0;     int[3] arr;      void_test(); // call param `Empty` test     // call param `void` test     //call param `expression` test     normal_test(void_test2(void)+3/9-4*arr[1]);     multi_args_test(1,2.0);      return a+3/9-4*arr[1]; //return `expression` test } </pre>	<ul style="list-style-type: none"> <li>▼ FuncDefStmt             <ul style="list-style-type: none"> <li>ReturnType: VOID</li> <li>ID : "void_test"</li> <li>FuncDefParams: VOID</li> <li>► InnerStmts</li> </ul> </li> <li>▼ FuncDefStmt             <ul style="list-style-type: none"> <li>► ReturnType</li> <li>ID : "void_test2"</li> <li>FuncDefParams: VOID</li> <li>► InnerStmts</li> </ul> </li> <li>▼ FuncDefStmt             <ul style="list-style-type: none"> <li>► ReturnType</li> <li>ID : "normal_test"</li> <li>► FuncDefParams</li> <li>► InnerStmts</li> </ul> </li> <li>▼ FuncDefStmt             <ul style="list-style-type: none"> <li>ReturnType: VOID</li> <li>ID : "multi_args_test"</li> <li>► FuncDefParams</li> <li>► InnerStmts</li> </ul> </li> <li>▼ FuncDefStmt             <ul style="list-style-type: none"> <li>► ReturnType</li> <li>ID : "main"</li> <li>FuncDefParams: VOID</li> <li>► InnerStmts</li> </ul> </li> </ul>
--	--



### (三)解释执行

#### 1. 算数运算

```
1 // test arithmetic
2
3 void main(){
4     int a=1;
5     write(a+2*3-(11-6)/2); // 1+6-5/2=7-2=5
6
7     real b=1.0;
8     write(a+2*3-(11-6.0)/2); // 1+6-5.0/2=7-2.5=4.5
9
10    return;
11 }
```

Console 

```
/home/finalize/Workspace/pycharm/cinter/test/5_execute/arith.t
5
4.5

Process finished successfully
```

#### 2. 数组

```
1 // test array
2
3 int[3] arr={1,2,3};
4
5 void main(){
6     int a=1;
7
8     arr[0]=1; // arr[0]:1
9     arr[a]=2; // arr[1]:2
10
11     a=arr[1];
12     arr[2]=a+1; // arr[2]:3
13
14     int i=0;
15     while(i<3){
16         write(arr[i]);
17         i=i+1;
18     }
19     return;
20 }
```

Console 

```
/home/finalize/Workspace/pycharm/cinter/test/5_execute/array.t
1
2
3

Process finished successfully
```

### 3. 条件判断

```
1 // test if-then-else-then if-then
2
3 void main(){
4     int a=9;
5
6     if(a>0){
7         write(a); // 9
8     }else{
9         a=0;
10    }
11
12    if(a==0){
13        write(a); // 0
14    }
15    return;
16 }
```

Console 

```
/home/finalize/Workspace/pycharm/cinter/test/5_execute/if.t
9
0

Process finished successfully
```

### 4. 循环

```
1 // test loop
2
3 void main(){
4     int a=10;
5
6     while(a>0){
7         write(a);
8         a=a-1;
9     }
10    return;
11 }
```

Console 


```
/home/finalize/Workspace/pycharm/cinter/test/5_execute/loop.t
10
9
8
7
6
5
4
3
2
1

Process finished successfully
```



## 5. 输入输出

```
1 void main() {
2     int a=read();
3     write(a);
4     return;
5 }
```

Console 

```
/home/finalize/Workspace/pycharm/cinter/test/5_execute/io.t
21321321321
21321321321

Process finished successfully
```

## 6. 函数

```
1 // test function call
2
3 int a;
4
5 int add(int a,int b){
6     return a+b;
7 }
8
9 real divide(real a,real b){
10     return a/b;
11 }
12
13 void print_int(int a){
14     write(a);
15     return;
16 }
```

Console 

```
/home/finalize/Workspace/pycharm/cinter/test/5_execute/func.t
19
5.0

Process finished successfully
```

## 7. 递归

```
1 int fibonacci(int n){
2     if(n==0){
3         return 0;
4     }
5     if(n<3){
6         return 1;
7     }
8
9     return fibonacci(n-1)+fibonacci(n-2);
10 }
11
12 void main(){
13     int i=0;
14     while(i<20){
15         int r=fibonacci(i);
16         write(r);
17         i=i+1;
18     }
19     return;
20 }
```

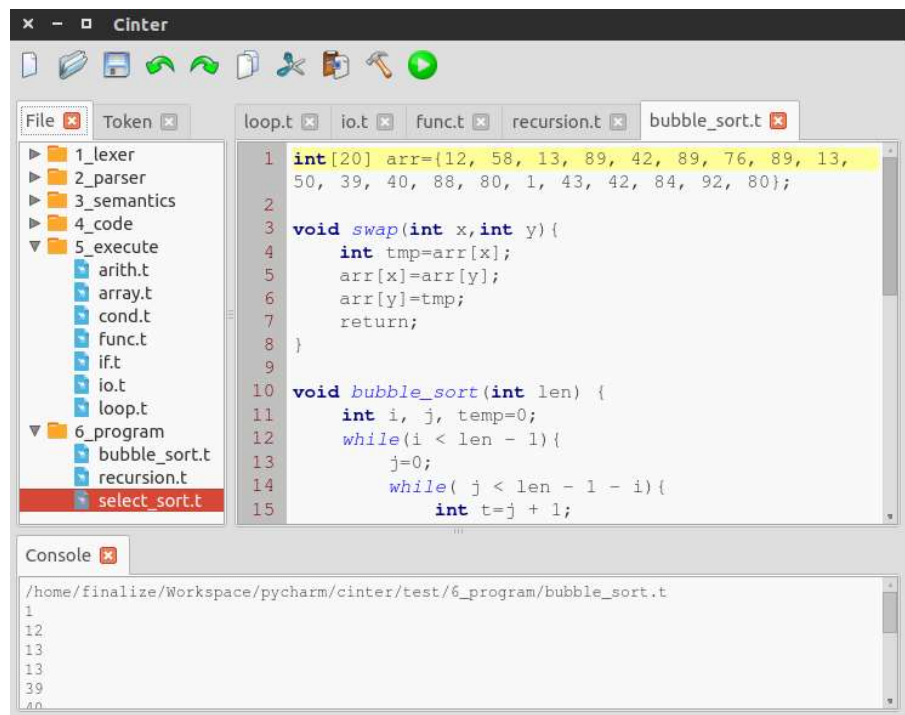
```
/home/finalize/Workspace/pycharm/cinter/test/6_program/recursion.t
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181

Process finished successfully
```

## 8. 冒泡排序

<pre>1 int[20] arr={12, 58, 13, 89, 42, 89, 76, 89, 13, 50, 39, 40, 88, 80, 1, 43, 42, 84, 92, 80}; 2 3 void swap(int x,int y){ 4     int tmp=arr[x]; 5     arr[x]=arr[y]; 6     arr[y]=tmp; 7     return; 8 } 9 10 void bubble_sort(int len) { 11     int i, j, temp=0; 12     while(i &lt; len - 1){ 13         j=0; 14         while( j &lt; len - 1 - i){ 15             int t=j + 1; 16             if (arr[j] &gt; arr[t]) { 17                 swap(j,j+1); 18             } 19             j=j+1; 20         } 21         i=i+1; 22     } 23     return; 24 } 25 26 void main() { 27     int len = 20; 28     bubble_sort(len); 29 30     int i=0; 31     while(i&lt;20){ 32         write(arr[i]); 33         i=i+1; 34     } 35     return; 36 }</pre>	<pre>/home/fina 1 12 13 13 39 40 42 42 43 50 58 76 80 80 84 88 89 89 89 92  Process fi</pre>
--	--

#### (四)界面



#### 六、 使用说明

执行环境： python2.7 PyQt5

执行方法： \$ python main.py

#### 七、 小结

1. 熟悉了解释器的编写流程，合理地应用了编译原理相关知识。
2. 增加了多模块项目的管理经验，以及调试经验。
3. 学会了使用 Qt GUI 库，掌握了信号和槽的基本原理
4. 错误处理机制还不够完善。
5. 自定义的语法太过于精简，导致一些常用的功能无法实现
6. 中间代码生成的指令太过于随意，应该采用与汇编类似的指令
7. 中间代码解释执行时偷懒，将函数参数放置到全局符号表，没有严格遵守函数堆栈的调用机制