1. PL0-compiler

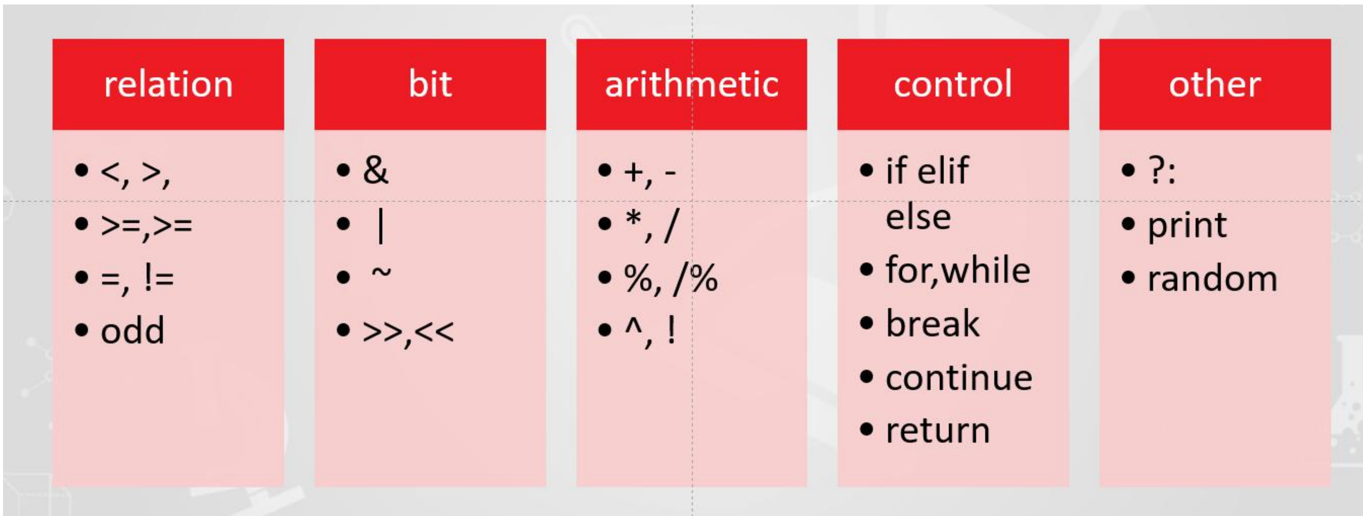
> A compiler for c-like programming language **based on** PL0, which is a dynamic, strong typing language.

> 小组成员: 朱河勤, 张世聪, 徐瑞, 詹慧悠

分工

- 朱河勤: 负责设计语法, 实现了 词法分析, 语法分析的 python 框架, 实现了代码生成与模拟,交互模式
- 张世聪: 负责调研资料,实现break, continue
- 徐瑞: 负责调研资料,实现 if elif else
- 詹慧悠: 负责调研资料, 实现 扩展的变量声明,

2. operations and features



3. Grammar

```
program =  body "."
body = {varDeclaration ";" |  constDeclaration ";" |  "func" ident "(" arg_list
")" body ";"}  sentence

varDeclaration = "var"  varIdent { "," varIdent}
varIdent  = ident ["=" number] | ident  { "[" number "]" }
constDeclaration = "const" ident "=" number {"," ident "=" number}

sentence = [ ident ":=" { ident ":=" } sentenceValue
                |  "begin" sentence { ";" sentence}  "end"
                |  "if" sentenceValue "then" sentence {"elif" sentence} ["else"
sentence]
                |  "while" sentenceValue "do" sentence
                |  "break"
                |  "continue"
```

```
                  |  ["return"] sentenceValue
                  |  "print" "(" real_arg_list ")" ]

sentenceValue =   condition

arg_list =  ident { "," ident}

real_arg_list = sentenceValue {"," sentenceValue }


condition = condition_or [ "?" sentenceValue ":" sentenceValue ]
condition_or  = condition_and { "||" condition_or }
condition_and = condition_not { condition_not "&&" condition_and}
condition_not = {"!"} condition_unit
condiiton_unit = ["odd"] expression
                        | expression ("<" | ">" | "<=" | ">=" | "=" | "!=")
expression

expression =  level1 { ("<<"| ">>" | "&" | "|") level1 }
level1  = level2 { ( "+" | "-" ) level2 }
level2 = level3 { "*" | "/" | "/%" | "%" ) level3 }
level3 = level4 {"^" level4}
level4 = item {"!"}          (*  factorial *)
item =  number  |ident { "(" real_arg_list ")" }| "(" sentenceValue" )" | ("+" |
"-" | "~" ) item
```

4. Instruction generation

We designed several instructions that can be generated for the target machine. To simplify this problem, we will emulate this virtual machine and execute instructions in python.

4.1. register

This machine has three registers:

- `b` is the base register that contains the base pointer to locate a varible in the data stack
- `reg` is the return register that contains the return value of latest function call
- `pc` is the pc register that points to the instruction

4.2. stack

There are two stack in this virtual machine. One contains the instructions, visited by register `pc`. It won't change when executing instructions, so we can assume it's readonly The other is data stack. It dynamiclly changes when running the program.

For each level, the first is the base address of this level. The second place is the static chain to visit the upper level's varibles. The third place contains the return address of the upper level. And the other places in one level contains local varibles and real time data for calculation.

Each time we call a function, the level increases 1. Also, the level decreases 1 when we return from a function.

4.3. instruction

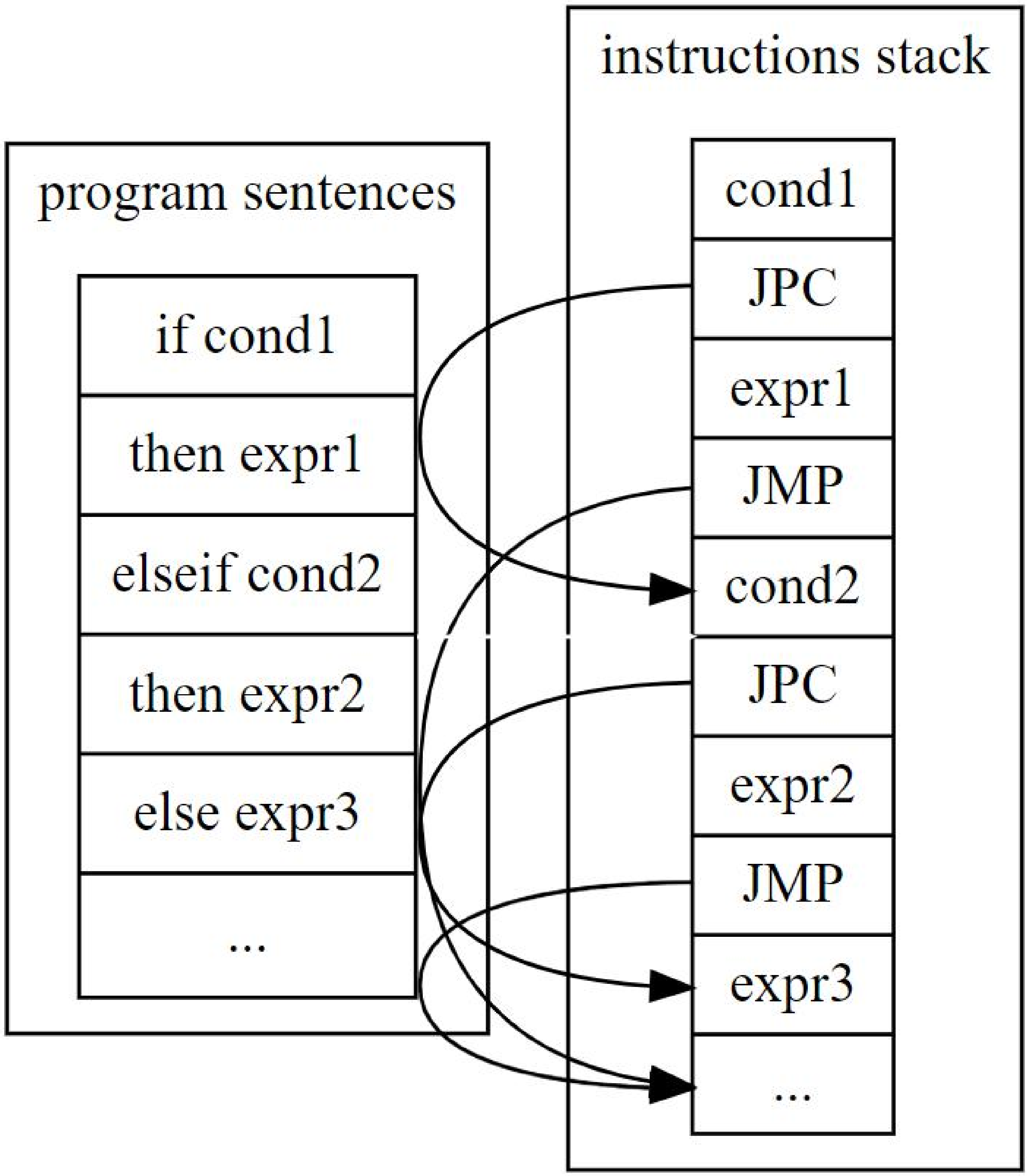
Every instruction consists of three parts. The first is the name of the instruction. Generally, the second is the level diifference of a identifier(if it has). And the third part is the address.

name	levelDiff	address	explanation
INT	-	n	allocate n space for one level
LIT	-	constant value	push a constant value to the top of the data stack
LOD	levelDiff	addr	load a varible value to the top of the data stack. The var can be found use levelDiff and addr
STO	levelDiff	addr	store the stack top value to a varible, top decreases.
CAL	levelDiff	addr	call a function
JMP	-	addr	jmp to addr, namely set addr to pc
JPC	-	addr	pop stack, if the value is not True, jmp addr
MOV	n1	n2	stk[top-n2] = stk[top-n1]
OPR	-	RET	return to the upper level, use current level's first three value to change pc, data stack, base register.
OPR	-	POP	pop the data stack, store the value in `reg` register
OPR	-	PUSH	push `reg` to stack top
OPR	n	BACK	rewind stk.top backward n steps
OPR	-	operator type	variout operation on value

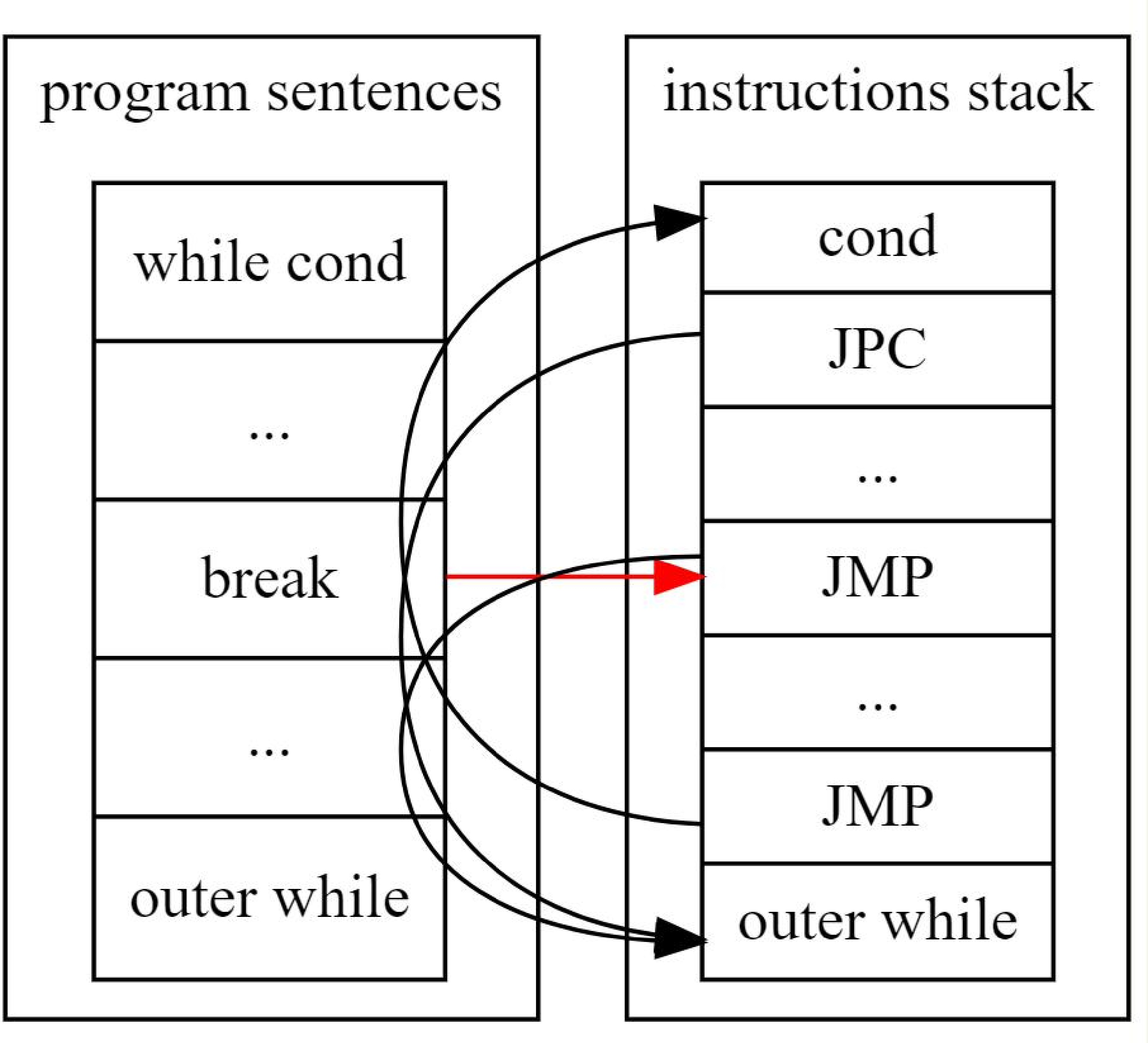
5. Design

We can generate instruction when analysing grammar. Some keypoints is the control structures' instruction traslation.

5.1. if elif else



5.2. while/break



`continue`, `for` can be translated in the same way.

5.3. function arguments pass

When analysing the function's defination, we can store the formal arguments as function's local varibles. As soon as we call this function, we should calculate the real arguments in the level upper the function, and then pass value to the function's formal varibles one by one.

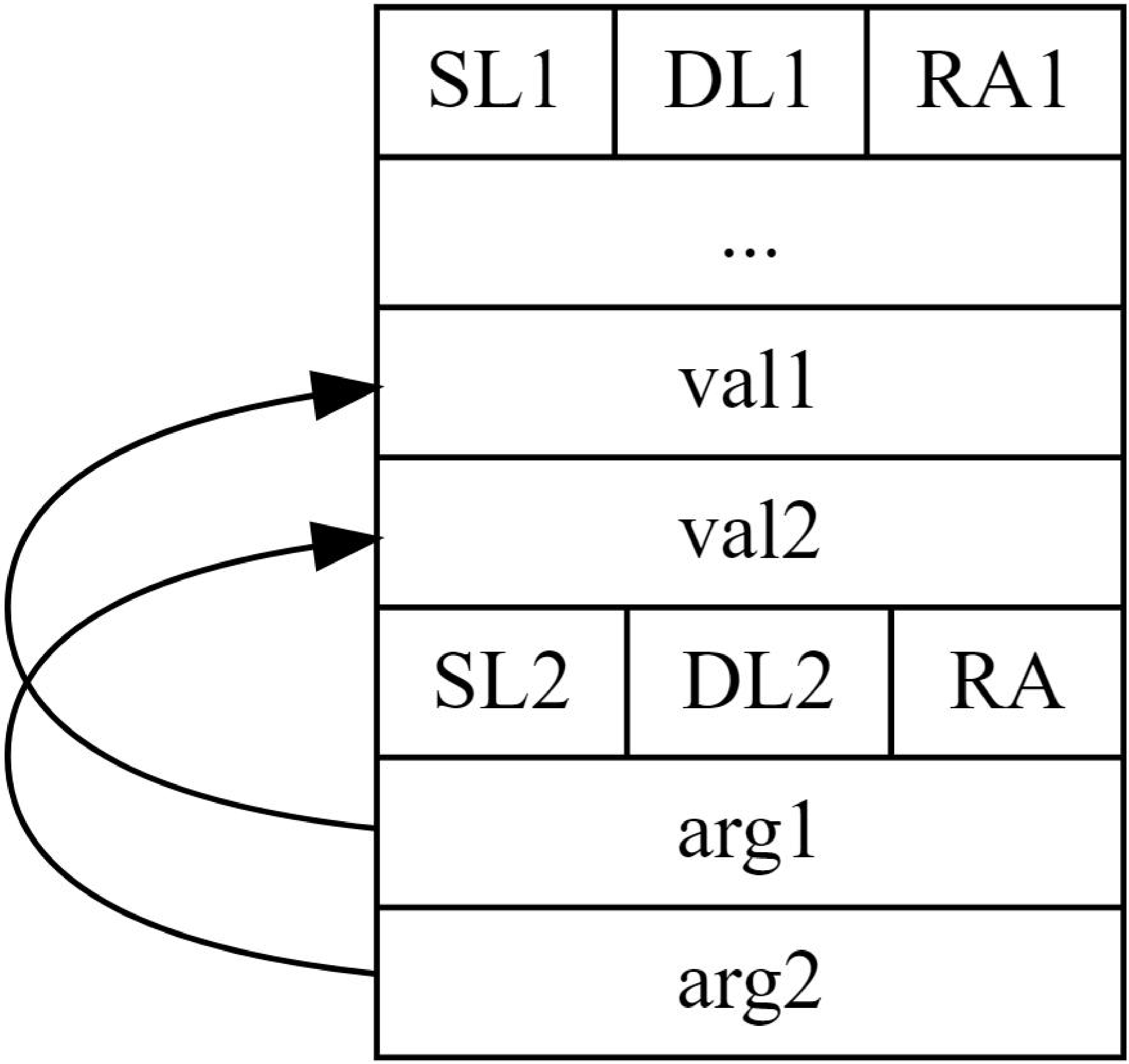
I use an instruction `MOV` to achive this goal. `MOV addr1, addr2` will store value stk[top-n2] in stk[top-n1]. Let's have a look at how to call a function and pass args value.

Before we call a function, its real args will be calculated in the level upper this function. Note function level is n+1, and we call this function in level n. In level n, we calculated function's args, all values are stored in the data stack of level n. Now call function and enter it. Data stack reaches level n+1 and grows three spaces for `DL`,`SL`,`RA`. The following space are for function's local varibles. So we can mov level n's real args value to these places according to function's argument num and varible num.

For example, function has n1 args, n2 local varibles(excluding args), then

```
for i in [0,1..,n1-1]:
    mov , n2+n1+3+i, n2 + i
```

The moment we returned level n, we should rewind top for n1 spaces, `OPR,n1,'BACK'` can make it.



5.4. function return

Also, mark function level as n+1, and outer(upper) is level n. To implement `return` sentence, we just need to do two things:

- calculate `return` sentence value **in level n+1**
- pass this value to level n It seems that it's hard to pass level n+1 's value to level n. Once we returned to level n, level n+1 's data in data stack will be cleared.

I use a extra register `reg` to achive this. Before we return,

- calculate return value
- `OPR ,0,'POP'` will pop the value and store it in reg
- return level n
- `OPR,0,'PUSH'` will push reg value to stack top

Now the return value has be passed from level n+1 to level n

5.5. instruction fillback

Taking `while` block as an example, Note that we don't know the `JPC` instruction's target addr until we finish analysing the whole block.The Solution is that after we analyse while condition, we generate an instruction with no target address, just take a place. We note down this instruction's address. As soon as we finish analysing the whole `while` block, the instruction pointer, namely `ip`, pointing to the target address of `JPC`. Then we fill back the `JPC` instruction with the target address along to ip.

5.6. symbol table

When analysing and translating, we want to get the symbol which including level, address,(value for constant) according to its name. The following shows how to achive it elegantly

There are three types of symbols:

- constant
- varible
- function name Every function has an environment that contains this level's symbols, and an outer environment(except main function). Every environment has the three symbols mentioned above.

Defaultly, we are in the main function in the beginning of this program.

In an enviroment, when we meet a symbol, we should seek it in current environment. If not found, go for the outer environment recursively until we found it.

It gurantees that every environment has no same names for different symbols but may have same names in different environment.

So there won't be conflits when different functions have same local varibles or arguments.

I create class `closure` to describe this kind of environment and varible `curClosure` to mark down current environment. Every time when calling a function, we enter a more inner environment. We do the following things to make sure that environment changes creately.

```
saved = curClosure
curClosure = function.closure
call function
curClosure = saved
```