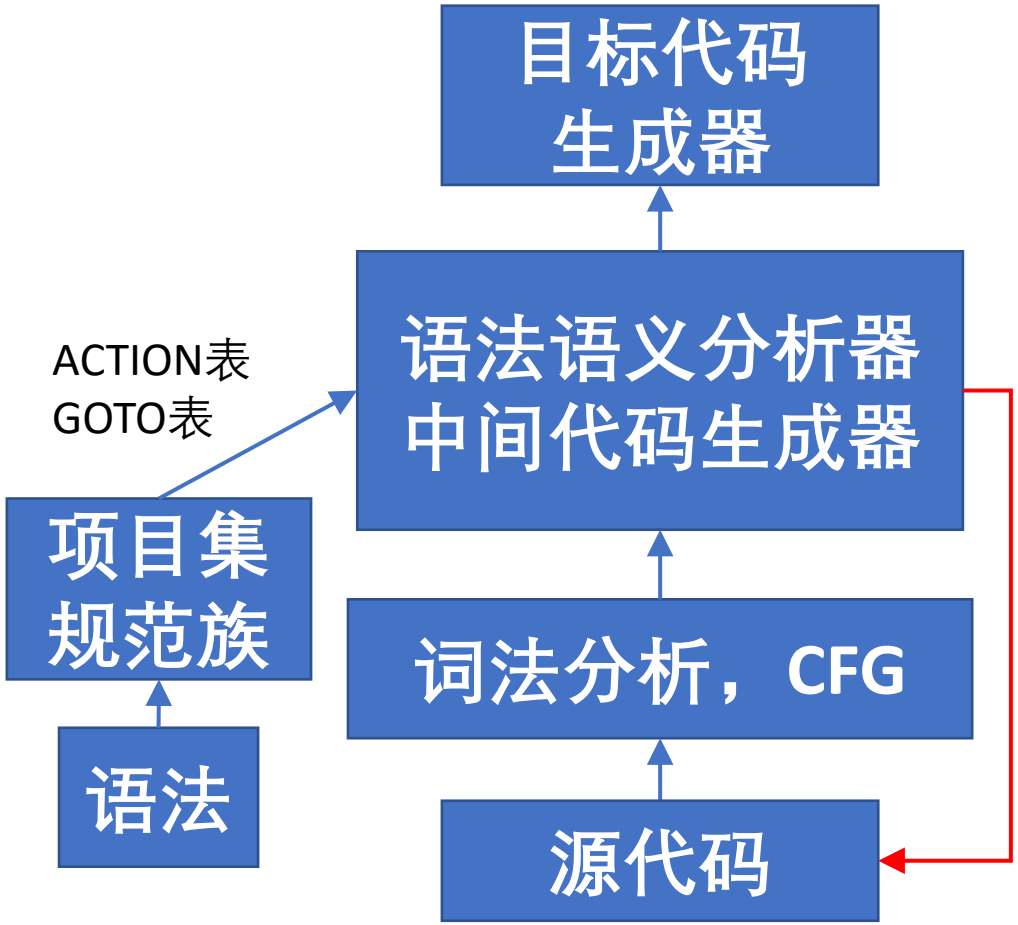
编译原理课设答辩
LR(1)编译器

1751740 刘鲲

需求分析： LR(1)编译器

- LR(1)在语法分析的同时生成中间代码，并保存到文件中
- 选用Python，减少代码量，但可能增加debug难度
- 带子程序调用
- 生成mips汇编代码
- 中间代码、汇编代码写入文件
- 词法分析程序作为子程序，需要的时候被语法分析程序调用

需求分析：系统模块框图



- 面向对象设计，方便维护和划分功能
- 语法、语义分析器、中间代码生成器合并
- 红色箭头：让词法分析作为子程序

CFG类

成员名称	功能描述
self.reserved	保留字
self.type	类别
self.regexs	词法分析用的正则表达式
getTokensOfOneLine	*对一行源代码作语法分析*
self.pInputStr	*记录语法分析位置的指针*

- Tokens是词法分析中间结果
- 以此实现“词法分析程序作为子程序，需要的时候被语法分析程序调用”

```
token['class'] = "T" # 变元/非变元
token['row'] = self.CURRENT_LINE
token['colum'] = origin.find(before)+1
token['name'] = self.type[self.regexs.i
token['data'] = result['data']
token['type'] = token['name']
```

主体代码展示说明

```
def getTokensOfOneLine(self, inputStr):
    # 从指针开始，得到下一行的位置
    idx = inputStr.find('\n', self.pInputStr)

    # 对这一行进行语法分析，返回tokens（语法分析中间结果）
    tokens = self.scan_line(inputStr[self.pInputStr:idx+1])

    # 更新指针
    self.pInputStr = idx + 1
    return tokens
```

ItemSetSpecificationFamily类

方法的名称	功能描述
self.itemSets	项目集，也即DFA状态
self.edges	DFA状态的边
getLR1Closure	根据LR1的方法算Closure集
GO	状态转移函数
getFirstSet	获取字符串的First集
extendItem	根据LR1文法将item进行终结符拓展
buildFamily	通过算法构建项目集族

SyntacticAnalyzer类(3合1:语法语义中间代码)

方法的名称	功能描述
getTables	计算ACTION和GOTO表
isRecognizable	判断一个字符串是否能被识别
semanticAnalyze (self, prod, shiftStr)	对特定产生式和tokens作语法分析+中间代码生成
saveMidCodeToFile	保存中间代码

```
1 def semanticAnalyze(self, prod, shiftStr):
2     if nt == 'program': ...
3     elif nt in ['statement',
4                 'block'
5                 ]:
6         TODO: 语义分析+中间代码生成
7     elif nt == 'declarationChain': ...
8     elif nt == 'typeSpecifier': ...
9     elif nt == 'declaration': ...
10      ...
```

isRecognizable

```
1 def isRecognizable(self, originCode):
2     inputStr = self.cfg.getTokensOfOneLine(originCode)
3     ...
4     while(True):
5         if len(inputStr) <= 2:
6             tmpInputStr = self.cfg.getTokensOfOneLine(originCode)
7             if len(tmpInputStr) == 0: # 如果已经读完了，加上"#"
8                 inputStr.append(wallSymbol)
9             else:
10                inputStr += tmpInputStr
11        mv = 从ACTION和GOTO表获取动作
12        if mv =='shift': ...
13        elif mv == 'goto': ...
14        elif mv == 'reduce': ...
15            # 规约时同时做中间代码生成
16            self.semanticAnalyze(prod, shiftStr)
17        elif mv == 'acc': ... return True
18        else: ... return False
```

输入串不够的时候调用词法分析

ObjectCodeGenerator类

属性名	描述
mipsCode	生成的Mips代码
regTable	记录此时寄存器内部存的是哪个变量
varStatus	记录变量此时是在寄存器/memory
getRegister	获取一个寄存器
freeRegister	释放寄存器
genMips	生成mips代码

过程调用目标代码生成设计

- 对于函数调用的语义分析，有多种四元式的翻译方法。
- 对于不同的四元式，有不同的目标代码翻译方法。
- 这两者最好一起设计!
- 目标代码：MIPS
- 设计方向/理想：不更改四元式的情况下可以更换成其它架构的目标代码生成器

过程调用

- 数据结构设计 & 给SyntacticAnalyzer新增方法

```
1 class FunctionSymbol:
2     def __init__(self):
3         self.name = None #函数的标识符
4         self.type = None #返回值类型
5         self.label = None #入口处的标签
6         self.params = [] #形参列表
7         self.tempVar = [] #局部变量列表
```

成员名	描述
updateFuncTable	更新函数表
getNewFuncLabel	获取函数符号
findFuncSymbolByName	由函数名获得函数符号

函数定义

```
# declareFunction -> typeSpecifier id ( formalParaList )
elif nt == 'declareFunction':
    f = FunctionSymbol() # 准备登记一个函数
    f.name = shiftStr[-4]['data'] # 函数名
    f.type = nFuncReturnType.type # 返回类型
    f.label = self.getNewFuncLabel() # 标签

    # 搜索formalParaList表，把参数列表记录下来
    for arg in formalParaList:
        s = Symbol()
        ...
    self.updateFuncTable(f)
```

```
# completeFunction -> declareFunction block
elif nt == 'completeFunction':
    code = []
    code.append(("函数名",':','_','_')) # 函数名

    for arg in argList:
        # 拿参数
```

函数调用主体代码

```
code=[]
# sp移动，申请空间，保存返回地址
code.append(('-', 'sp', 4 * len(symbol_temp_list) + 4, 'sp')) # 注意+4给ra留空间
code.append(('store', '_', 4 * len(symbol_temp_list), 'ra')) # 保存ra的值避免复写

for node in n.stack: # 实参列表
    code.append(('push','_',4 * n.stack.index(node),node_result))

code.append(('call', '_', '_', function.label))

# 恢复现场
code.append(('load', '_', 4 * len(symbol_temp_list), 'ra'))
code.append(('+', 'sp', 4 * len(self.curFuncSymbol.params) + 4, 'sp'))

# 返回值在v0寄存器里
code.append((':=', 'v0', '_', n.place))
```

目标代码

- elif code[0] == 'call':
- 	objCode.append('jal {}'.format(code[3]))

- 其它的目标代码和四元式差不多

自己设定的语法是否合理?

- 语法过于复杂的情况下，人工去判断是否是LR（1）语法（算follow集）基本不可能（18个非终结符，130个状态）。
- 语法本身设计的错误，体现在生成ACTION和GOTO表的时候会复写

```
if self.ACTION[I.name][item.terms[0]] != ' ':
    print('rewrite error!!!')
```

- 往往是在语义分析的时候才知道原来设计的语法质量，好的语法设计是为语义分析省事。通过这种方式查错、优化和消除继承属性，快速迭代出合理的语法

递归产生式

- declarationChain -> $ |declaration declarationChain
- statementChain -> statement | statementChain statement
- paraList -> para | paraList , para
- 使用栈存储结果
- 在这些产生式的左部出现在其它产生式的右边时才翻译成中间代码
- 优化：对于每一条产生式都要配语义规则，越少的产生式工作量越少

```
1 class Node():
2     def __init__(self):
3         ...
4         self.stack = [] #翻译闭包表达式所用的临时
5 栈       ...
```

函数语法设计

```
completeFunction
    declareFunction block
    #
```

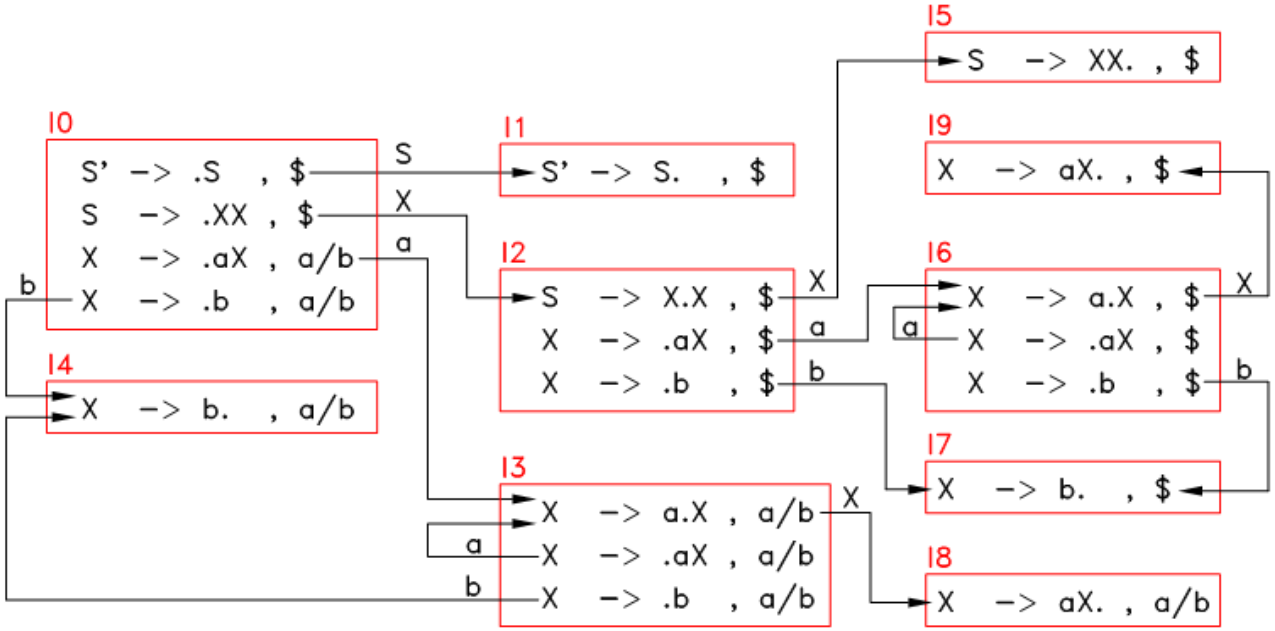
```
declareFunction
    typeSpecifier id ( formalParaList )
    #
```

```
block
    { statementChain }
    #
```

- 为什么不一个产生式完成函数定义，非要分三块呢？
- 当我们处理statementChain时，我们需要为里面的临时变量进行登记
- 但如果我们是在一条产生式中完成整个函数的定义，由自底向上文法的规则，就会发现在处理语句块的时候，函数还没有登记过。
- 所以将函数头声明的语句先完成，赋予函数的登记的语义规则，那么接下来的语句块的变量等等都有地方登记。

有空产生式怎么办?

- GO函数不把空串当作任何字符（NT/T）处理
- A->.ε和A->ε.等价于A->.
- 最后reduce的时候，规约栈要压入A，状态栈压入GO(I,A)所到达的状态



存在左递归时的First集求解

- A -> Aa， First(A)?
- First(A) += First(A)
- **无贡献：不一定！**
- **死循环：及时退出/跳过**

- 对于所有NT，计算First?
- for nt in NonTerminal:
  - calNTFirstSet(nt)
- 错误写法!

再考虑两种情况

- A->epsilon
- A->AB ->能够拿到First(B)

或

- A->AB
- A->epsilon ->错过First(B)

- 顺序遍历产生式时就可能会错过之前的左递归
- 为完备，必然要多次遍历，不是O(n)

正确做法

- 并发计算first集：按照笔算的方法，**能算出几个字符，每个字符算出几个，就算几个**，直到不再增加
- While(1):
  - isBigger =0
  - 计算first集
  - if isBigger == 1:
    - Break

在将某元素加入到某个集合中，判断是否该元素已存在该集合中

1.给item写一个string方法（哈希函数）

2.然后每添加一个item到一个itemSet，就用Python字典记录这个string：

```
setStrings[tempItemSet.name] = tempItemSet.toString()
```

3.每次要加入新的item时，就用它的string在记录中比对：

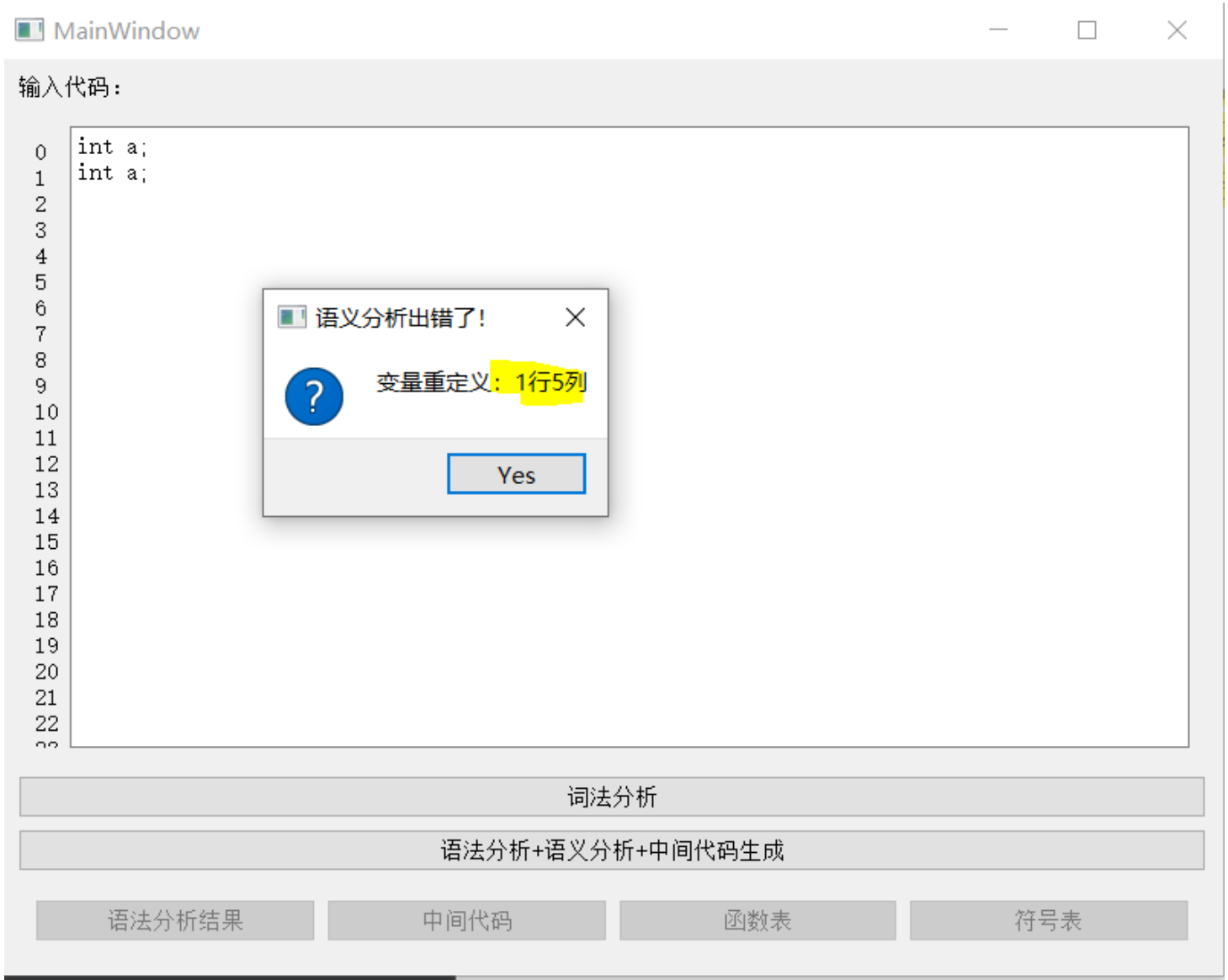
```
if tempItemSet.toString() in setStrings.values():
```

错误提示

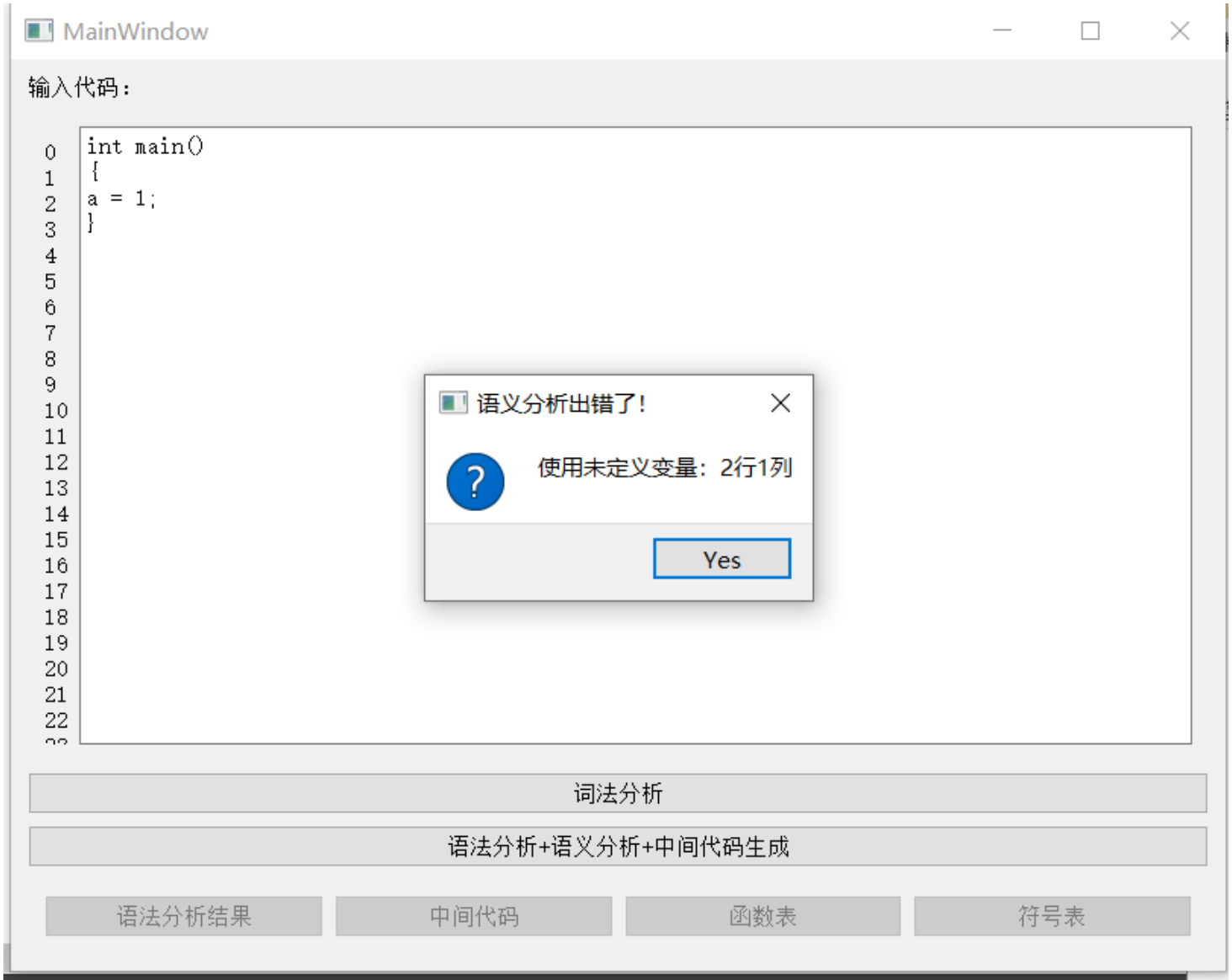
- 变量重定义
- 使用未声明的变量
- 使用未定义的函数
- 变量赋值时类型错误
- 函数形参和实参不匹配

```
token['class'] = "T" # 变元/非变元
token['row'] = self.CURRENT_LINE
token['colum'] = origin.find(before)+1
token['name'] = self.type[self.regexs.i
token['data'] = result['data']
token['type'] = token['name']
```

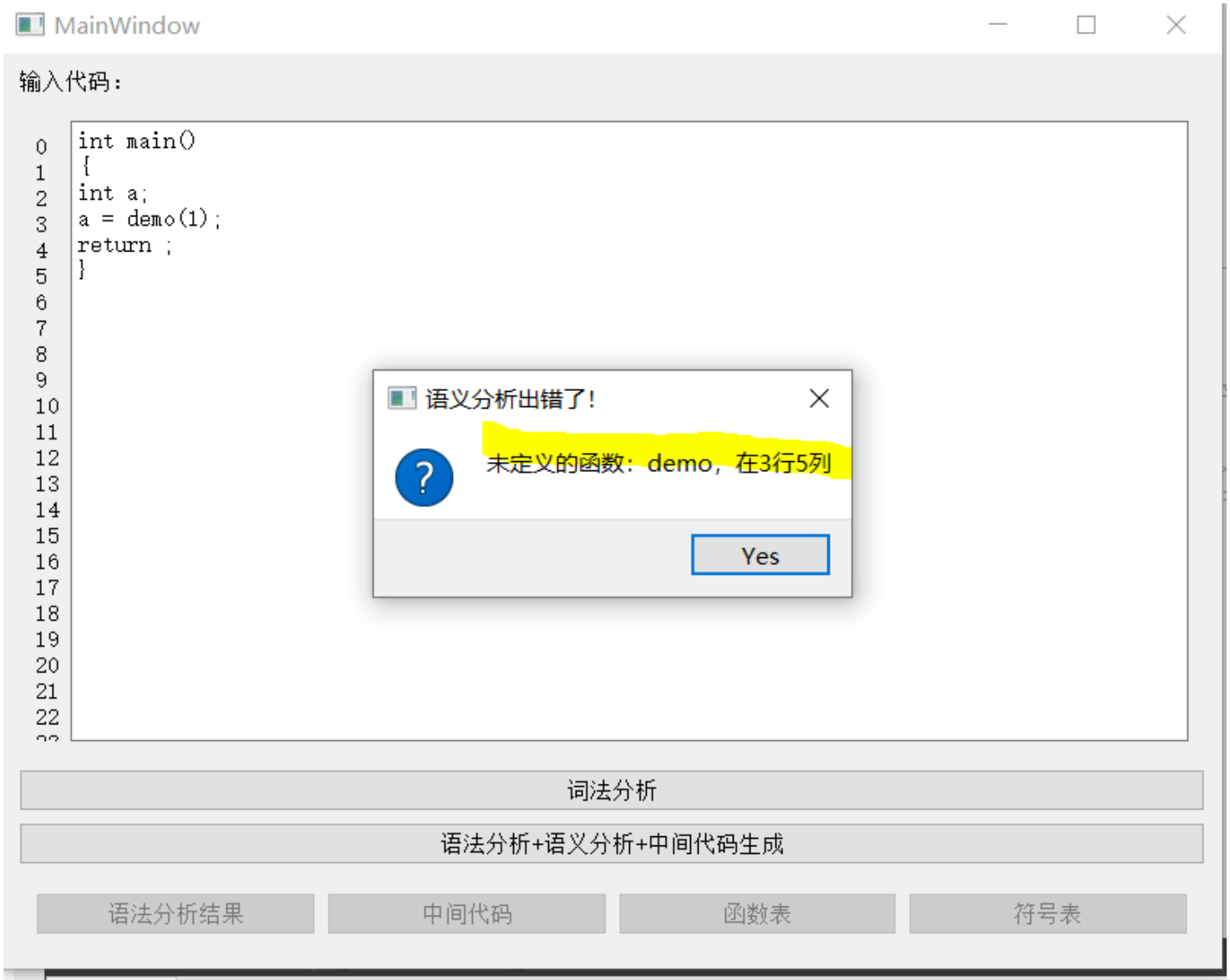
变量重定义


MainWindow
输入代码：
int a;
int a;
语义分析出错了！
变量重定义：1行5列
Yes
词法分析
语法分析+语义分析+中间代码生成
语法分析结果
中间代码
函数表
符号表

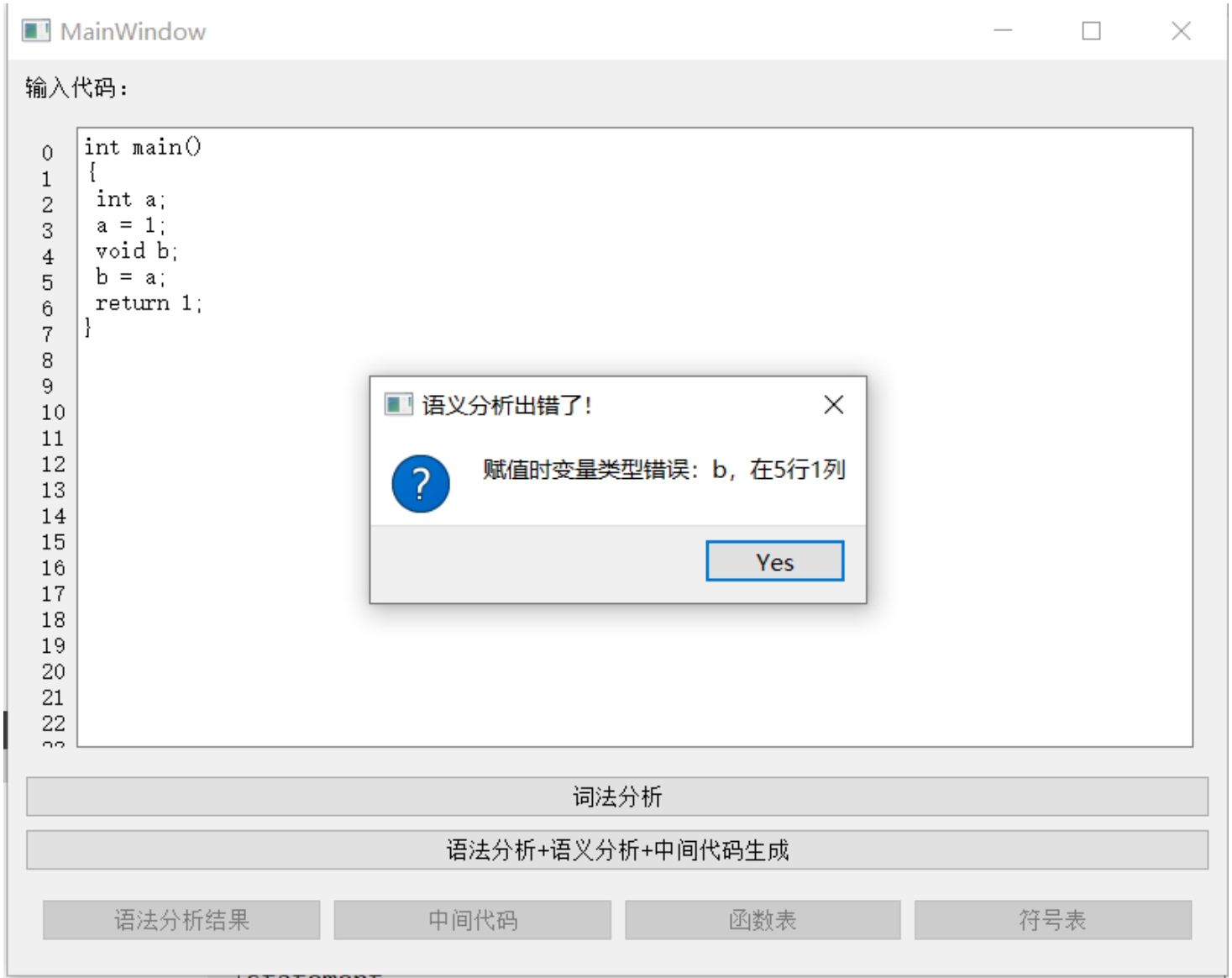
使用未声明的变量


MainWindow
输入代码：
int main()
{
a = 1;
}
语义分析出错了！
使用未定义变量：2行1列
Yes
词法分析
语法分析+语义分析+中间代码生成
语法分析结果
中间代码
函数表
符号表

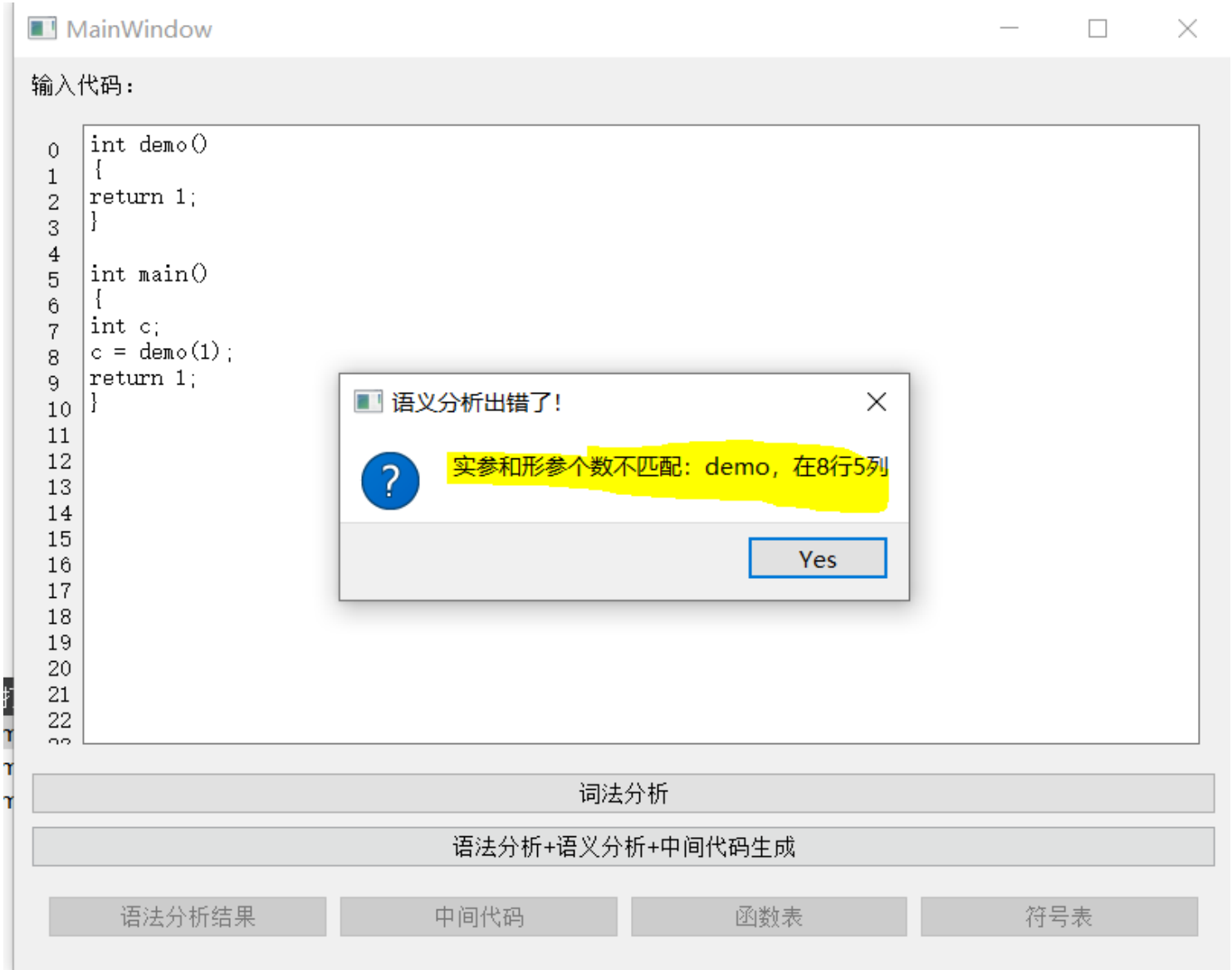
使用未定义的函数


MainWindow
输入代码：
0 int main()
1 {
2 int a;
3 a = demo(1);
4 return ;
5 }
语义分析出错了！
未定义的函数：demo，在3行5列
Yes
词法分析
语法分析+语义分析+中间代码生成
语法分析结果
中间代码
函数表
符号表

变量赋值时类型错误


MainWindow
输入代码：
int main()
{
int a;
a = 1;
void b;
b = a;
return 1;
}
语义分析出错了！
赋值时变量类型错误：b，在5行1列
Yes
词法分析
语法分析+语义分析+中间代码生成
语法分析结果
中间代码
函数表
符号表

函数形参和实参不匹配


MainWindow
输入代码：
0 int demo()
1 {
2 return 1;
3 }
4
5 int main()
6 {
7 int c;
8 c = demo(1);
9 return 1;
10 }
语义分析出错了！
实参和形参个数不匹配：demo，在8行5列
Yes
词法分析
语法分析+语义分析+中间代码生成
语法分析结果
中间代码
函数表
符号表

寄存器取用测试

```
int main(void)
{
    int a;int b;int c;int d;int e;int f;int g;int h;int i;int j;int k;int l;int m;int n;int o;
    int p;int q;int r;int s;int t;int u;

    a=0;b=1;c=2;d=3;e=4;f=5;g=6;h=7;i=8;j=9;k=10;l=11;m=12;n=13;o=14;p=15;q=16;r=17;s=18;t = 19; u = 20;
    a=0;b=1;c=2;d=3;e=4;f=5;g=6;h=7;i=8;j=9;k=10;l=11;m=12;n=13;o=14;p=15;q=16;r=17;s=18;t=19;

    return 1;
}
```

```
addiu $sp, $zero, 0x10018000
or $fp, $sp, $zero
jal  main
jal  programEnd
main:
add $7,$zero,0
add $8,$zero,1
add $9,$zero,2
add $10,$zero,3
add $11,$zero,4
add $12,$zero,5
add $13,$zero,6
add $14,$zero,7
add $15,$zero,8
add $16,$zero,9
add $17,$zero,10
add $18,$zero,11
add $19,$zero,12
add $20,$zero,13
add $21,$zero,14
add $22,$zero,15
add $23,$zero,16
add $24,$zero,17
add $25,$zero,18
addi $at, $zero, 0x10010000
sw $7, 0($at)
add $7,$zero,19
addi $at, $zero, 0x10010000
sw $8, 4($at)
add $7,$zero,20
add $8,$zero,0
add $8,$zero,1
add $9,$zero,2
```

演示视频

感谢聆听！