

课程报告

一、编译程序概述

1.功能

编译程序能够将高级语言代码转换为计算机可执行的机器语言代码。它由多个组件组成，包括词法分析器、语法分析器、语义分析器和中间代码生成器。

首先，编译程序的第一个任务是通过词法分析器将源代码转换为标记流（Token Stream）。词法分析器将源代码分割成一个个标记，每个标记由<种别编码，单词符号的属性值>组成。在这个过程中，词法分析器还会删除注释和空格等无用信息，使得源代码更加精简和易于处理。

接下来，语法分析器接收词法分析器生成的标记流，并将其转换为抽象语法树（Abstract Syntax Tree, AST）。AST是源代码的一种层次化表示方式，其中每个节点表示源代码中的一个构造或操作。语法分析器通过分析标记流的语法结构，构建起一个清晰的抽象语法树，反映出源代码的逻辑结构和语法关系。

最后，中间代码生成器使用语义分析器生成的语义信息，将抽象语法树转换为中间代码。中间代码是一种介于源代码和目标机器代码之间的中间表示形式，它更接近机器语言，但仍保留了一定的抽象性。中间代码生成器根据目标平台的要求，将抽象语法树转化为适合目标平台执行的中间代码。

2.编译程序结构

词法分析器模块：将源代码转换成标记流，包含种别编码和属性值。

语法分析器模块：根据标记流构建抽象语法树，反映源代码的结构和关系。

语义分析和中间代码生成模块：检查语义正确性，并生成适合目标平台的中间代码。

错误处理模块：检测和处理编译过程中的错误，提供错误信息和修复建议。

符号表模块：存储标识符及其属性，支持作用域嵌套和解析，确保标识符引用的正确性。

二、词法分析

1.功能

识别单词模块：将输入的字符流转换为具有特定含义的单词或词法单元，确定其类别和属性，为后续的语法分析提供基础。

去除空格模块：消除代码中的空格、制表符、换行符和注释等无关内容，提供一个简洁的代码流用于后续的词法和语法分析。

生成目标二元组模块：将源代码转换为目标二元组，为后续的语法分析提供基础。

2.实现

```
def tokenize(code, index):
    operators = ['++', '--', '+', '-', '*', '/', '%', '=', '>', '>=', '<', '<=',
                '==', '!=', '&&', '||', '!']
    delimiters = [',', '(', ')', ';', '{', '}', '[', ']']
    keywords = {'main', 'int', 'char', 'if', 'else', 'for', 'while', 'return',
                'void'}
    i = index

    while i < len(code):
        if code[i].isspace() or code[i] == '\n':
            return ('', i + 1) # 返回空标记和更新后的索引

        if code[i:i + 2] in operators:
            return (code[i:i + 2], i + 2) # 返回运算符标记和更新后的索引

        if code[i] in operators:
            return (code[i], i + 1) # 返回运算符标记和更新后的索引

        if code[i] in delimiters:
            return (code[i], i + 1) # 返回分隔符标记和更新后的索引

        if code[i].isalpha() or code[i] == '_':
            j = i + 1
            while j < len(code) and (code[j].isalnum() or code[j] == '_'):
                j += 1
            token = code[i:j]
            if token in keywords:
                return (token, j) # 返回关键字标记和更新后的索引
            else:
                return (token, j) # 返回标识符标记和更新后的索引

        if code[i].isdigit():
            j = i + 1
            while j < len(code) and code[j].isdigit():
                j += 1
            return (code[i:j], j) # 返回数字标记和更新后的索引

    return ('', i) # 返回空标记和当前索引
```

上述代码实现了一个用于词法分析的函数 `tokenize(code, index)`。函数接受两个参数，一个是待分析的代码字符串 `code`，另一个是分析的起始位置 `index`。

在函数内部，首先定义了操作符、分隔符和关键字的列表和集合。然后通过一个循环遍历代码字符串，从给定的索引位置开始逐个字符进行分析。

循环中的每个条件判断均用于确定当前字符或字符组合的类型，并返回相应的标记和更新后的索引。具体判断逻辑如下：

1. 如果当前字符为空格或换行符，则返回空标记和更新后的索引。

2. 如果当前字符及其后一个字符是操作符，则返回运算符标记和更新后的索引。
3. 如果当前字符是单个操作符，则返回运算符标记和更新后的索引。
4. 如果当前字符是分隔符，则返回分隔符标记和更新后的索引。
5. 如果当前字符是字母或下划线，从当前位置开始向后遍历，直到遇到非字母、非数字和非下划线字符，得到一个标识符或关键字，并返回相应的标记和更新后的索引。
6. 如果当前字符是数字，从当前位置开始向后遍历，直到遇到非数字字符，得到一个数字标记，并返回标记和更新后的索引。

如果循环结束后仍未匹配任何条件，则返回空标记和当前索引。

该函数逐个字符扫描代码字符串，根据字符的类型返回相应的标记，可以用于将代码字符串划分为不同的词法单元，方便后续的语法分析和编译过程。

```
def get_token(line_num, col_num, lines):
    if line_num == len(lines):
        return ('%%done', -1, -1)

    (token, col_num) = tokenize(lines[line_num], col_num)

    if token == '':
        if col_num < len(lines[line_num]):
            return get_token(line_num, col_num, lines)
        else:
            return get_token(line_num + 1, 0, lines)
    else:
        if col_num < len(lines[line_num]):
            return (token, line_num, col_num)
        else:
            return (token, line_num + 1, 0)
```

该代码实现了一个用于获取词法单元的函数 `get_token(line_num, col_num, lines)`。函数接受三个参数，`line_num` 表示当前行号，`col_num` 表示当前列号，`lines` 是包含代码行的列表。

函数首先判断当前行号是否等于 `lines` 列表的长度，如果是，则表示已经扫描完所有的行，返回特殊的标记 `('%%done', -1, -1)`，表示词法分析结束。

如果当前行号小于 `lines` 列表的长度，那么就调用 `tokenize` 函数对当前行的代码进行词法分析。将返回的标记和更新后的列号存储在 `(token, col_num)` 元组中。

接下来进行条件判断：

1. 如果标记为空，则表示当前行的代码已经全部分析完毕，需要判断是否还有下一行。如果当前列号小于当前行的长度，则递归调用 `get_token` 函数，并将行号和列号作为参数传递，继续分析当前行。否则，表示当前行已经全部分析完毕，需要分析下一行，因此递归调用 `get_token` 函数，行号加1，列号置为0，继续分析下一行的代码。
2. 如果标记不为空，表示当前行的代码还有未分析的部分。如果当前列号小于当前行的长度，则返回标记、行号和列号作为结果。否则，表示当前行的代码已经全部分析完毕，需要分析下一行，因此返回标记、行号加1，列号置为0，作为结果。

```
def advance():
    global line_num, col_num
    (token, line_num, col_num) = get_token(line_num, col_num, lines)
```

这段代码的目的是将全局变量 `line_num` 和 `col_num` 的值更新为 `get_token()` 函数返回的相应值，实现了在代码解析过程中逐步前进的功能。通过调用 `advance()` 函数，可以更新全局变量的值，以获取下一个词法单元。

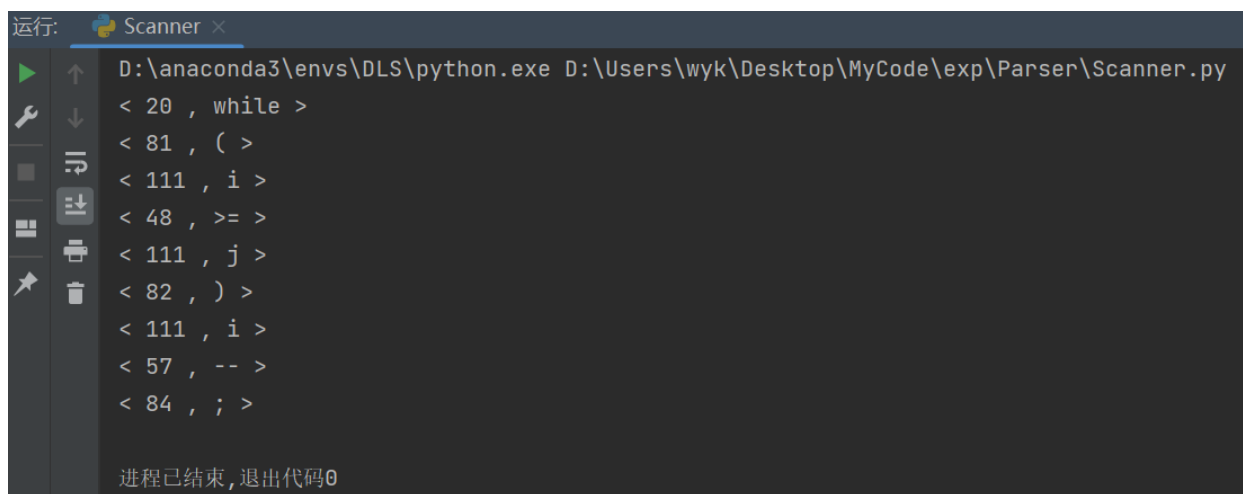
3.测试

输入

```
while(i>=j) i--;
```

输出

```
< 20 , while >
< 81 , ( >
< 111 , i >
< 48 , >= >
< 111 , j >
< 82 , ) >
< 111 , i >
< 57 , -- >
< 84 , ; >
```



```
运行: Scanner x
D:\anaconda3\envs\DLS\python.exe D:\Users\wyk\Desktop\MyCode\exp\Parser\Scanner.py
< 20 , while >
< 81 , ( >
< 111 , i >
< 48 , >= >
< 111 , j >
< 82 , ) >
< 111 , i >
< 57 , -- >
< 84 , ; >

进程已结束,退出代码0
```

三、语法分析

1.功能

识别语法结构模块：用于分析输入的代码，检查语法是否符合规定的语法结构，识别和验证各种语法规则的出现和组合。

生成抽象语法树模块：将经过语法分析的代码转换为一棵树形结构，其中每个节点表示代码的一个语法结构，例如表达式、语句、函数定义等。这个树形结构可以更方便地表示代码的结构和层次关系，为后续的语义分析和代码生成提供基础。

2.实现

以文法stmt为例

```
stmt -> if(bool) stmt else stmt | while(bool) stmt | loc = expr;
```

实现如下

```
def stmt():
    global line_num, col_num
    (token, line_num_, col_num_) = get_token(line_num, col_num, lines)
    if token == 'if':
        print("stmt -> if(bool) stmt else stmt")
        advance()
        advance()
        bool()
        advance()
        stmt()
        advance()
        stmt()
    elif token == 'while':
        print("stmt -> while(bool) stmt")
        advance()
        advance()
        bool()
        advance()
        stmt()
    elif judge_token(token) == 111:
        print("stmt -> loc = expr;")
        loc()
        advance()
        expr()
        advance()
```

函数首先调用 `get_token(line_num, col_num, lines)` 函数，将返回的结果存储在 `(token, line_num_, col_num_)` 元组中。

然后根据 `token` 的值进行条件判断，判断当前语句的类型。

- 如果 `token` 等于 `'if'`，表示当前语句是一个条件语句，打印输出对应的产生式，并依次调用 `advance()` 函数进行词法单元的前进、`bool()` 函数分析布尔表达式、`stmt()` 函数分析条件为真时的语句块、再次调用 `advance()` 函数前进、最后调用 `stmt()` 函数分析条件为假时的语句块。

- 如果 token 等于 'while'，表示当前语句是一个循环语句，打印输出对应的产生式，并依次调用 advance() 函数进行词法单元的前进、bool() 函数分析循环条件、再次调用 advance() 函数前进、最后调用 stmt() 函数分析循环体的语句块。
- 如果 judge_token(token) == 111，表示当前语句是一个赋值语句，打印输出对应的产生式，并依次调用 loc() 函数分析左侧变量、再次调用 advance() 函数前进、expr() 函数分析右侧表达式、最后调用 advance() 函数前进。

该函数根据当前语句的类型，打印对应的产生式，并调用其他函数进行语法分析。通过递归调用和条件判断，实现了逐步解析语句的语法结构。

3.测试

输入

```
while(sum<10000)
    if(a<b)
        sum=sum*(c[10]+10);
    else
        c[10]=sum*c[10]+10;
x[i,j]=sum;
```

输出

```
stmts -> stmt rest0
stmt -> while(bool) stmt
bool -> equality
equality -> rel rest4
rel -> expr rop_expr
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> loc
loc -> id resta
resta -> ε
rest6 -> ε
rest5 -> ε
rop_expr -> <expr
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> num
rest6 -> ε
rest5 -> ε
rest4 -> ε
stmt -> if(bool) stmt else stmt
bool -> equality
equality -> rel rest4
rel -> expr rop_expr
expr -> term rest5
term -> unary rest6
```

```
unary -> factor
factor -> loc
loc -> id resta
resta -> ε
rest6 -> ε
rest5 -> ε
rop_expr -> <expr
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> loc
loc -> id resta
resta -> ε
rest6 -> ε
rest5 -> ε
rest4 -> ε
stmt -> loc = expr;
loc -> id resta
resta -> ε
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> loc
loc -> id resta
resta -> ε
rest6 -> *unary rest6
unary -> factor
factor -> (expr)
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> loc
loc -> id resta
resta -> [elist]
elist -> expr rest1
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> num
rest6 -> ε
rest5 -> ε
rest1 -> ε
rest6 -> ε
rest5 -> +term rest5
term -> unary rest6
unary -> factor
factor -> num
rest6 -> ε
rest5 -> ε
rest6 -> ε
rest5 -> ε
stmt -> loc = expr;
```

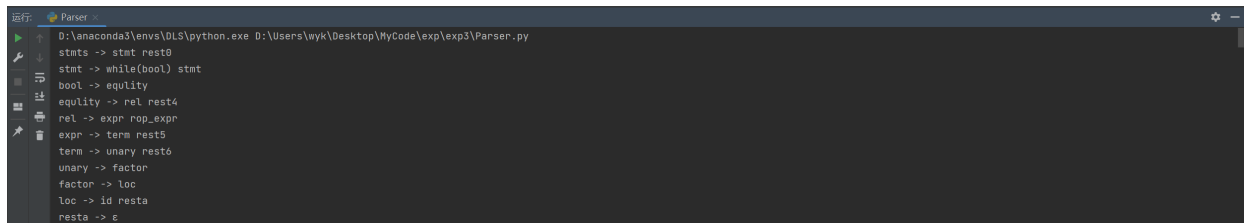
```
loc -> id resta
resta -> [elist]
elist -> expr rest1
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> num
rest6 -> ε
rest5 -> ε
rest1 -> ε
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> loc
loc -> id resta
resta -> ε
rest6 -> *unary rest6
unary -> factor
factor -> loc
loc -> id resta
resta -> [elist]
elist -> expr rest1
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> num
rest6 -> ε
rest5 -> ε
rest1 -> ε
rest6 -> ε
rest5 -> +term rest5
term -> unary rest6
unary -> factor
factor -> num
rest6 -> ε
rest5 -> ε
rest0 -> stmt rest0
stmt -> loc = expr;
loc -> id resta
resta -> [elist]
elist -> expr rest1
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> loc
loc -> id resta
resta -> ε
rest6 -> ε
rest5 -> ε
rest1 -> ,expr rest1
expr -> term rest5
term -> unary rest6
```



```

unary -> factor
factor -> loc
loc -> id resta
resta -> ε
rest6 -> ε
rest5 -> ε
rest1 -> ε
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> loc
loc -> id resta
resta -> ε
rest6 -> ε
rest5 -> ε
rest0 -> ε

```



```

D:\anaconda3\envs\DL5\python.exe D:\Users\wyk\Desktop\MyCode\exp\exp3\Parser.py
stmts -> stmt rest0
stmt -> while(bool) stmt
bool -> equality
equality -> rel rest4
rel -> expr rop_expr
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> loc
loc -> id resta
resta -> ε

```

四、语义分析和中间代码生成

1.功能

语义分析模块：对经过语法分析的代码进行语义检查，验证代码的合法性、类型匹配和语义规则等，捕捉和报告潜在的语义错误。

生成中间代码模块：将经过语义分析的代码转换为中间表示形式，该形式便于进一步的优化和目标代码生成。

2.实现

(1)赋值语句的翻译

```

def emit(op, arg1, arg2, result):
    quadruple = (op, arg1, arg2, result)
    quadruples.append(quadruple)

def newtemp():
    global temp_count
    temp_var = "t" + str(temp_count)
    temp_count += 1
    return temp_var

```

`emit(op, arg1, arg2, result)` 函数用于生成中间代码的四元组 (quadruple)，接收操作符 `op`、参数 `arg1`、`arg2` 和结果 `result` 作为输入。函数内部创建一个四元组 `quadruple`，将输入的值按顺序存储在该四元组中，然后将该四元组添加到 `quadruples` 列表中。

`newtemp()` 函数用于生成临时变量名，该临时变量名采用 "t" 加上递增的 `temp_count` 的形式。函数内部首先声明全局变量 `temp_count`，然后创建一个临时变量名 `temp_var`，将其设为 "t" 加上 `temp_count` 的字符串形式。接着将 `temp_count` 递增 1，并返回临时变量名 `temp_var` 作为结果。

以文法 `rest6` 为例

```
rest6 -> *unary rest6 | /unary rest6 | ε
```

实现如下

```
def rest6(rest6_in):
    global line_num, col_num
    (token, line_num_, col_num_) = get_token(line_num, col_num, lines)
    if token == '*':
        advance()
        unary_place = unary()
        rest61_in = newtemp()
        emit('*', rest6_in, unary_place, rest61_in)
        return rest6(rest61_in)
    elif token == '/':
        advance()
        unary_place = unary()
        rest61_in = newtemp()
        emit('/', rest6_in, unary_place, rest61_in)
        return rest6(rest61_in)
    else:
        return rest6_in
```

函数首先调用 `get_token(line_num, col_num, lines)` 函数，将返回的结果存储在 `(token, line_num_, col_num_)` 元组中。

然后根据 `token` 的值进行条件判断，判断当前的运算符是乘法还是除法。

- 如果 `token` 等于 `*`，表示当前是乘法运算，调用 `advance()` 函数前进一个词法单元，然后调用 `unary()` 函数获取乘法运算的右操作数，创建一个新的临时变量名 `rest61_in`，使用 `emit('*')` 生成中间代码将左操作数、右操作数和临时变量名作为参数，然后递归调用 `rest6(rest61_in)` 处理剩余部分。
- 如果 `token` 等于 `/`，表示当前是除法运算，与乘法运算类似，执行相同的操作流程。
- 如果不满足以上两个条件，即为其他情况，直接返回 `rest6_in` 作为结果。

(2)数组的翻译

以文法elist为例

```
elist -> expr rest1
```

实现如下

```
def elist(elist_inArray):  
    expr_place = expr()  
    rest1_inArray = elist_inArray  
    rest1_inNidm = 1  
    rest1_inPlace = expr_place  
    (elist_array,elist_offset) = rest1(rest1_inArray,rest1_inNidm,rest1_inPlace)  
    return(elist_array,elist_offset)
```

elist() 函数用于处理表达式列表的语法结构。函数首先调用 expr() 函数，将返回的结果存储在变量 expr_place 中，表示表达式的值。接下来，函数将一些参数赋值给变量 rest1_inArray、rest1_inNidm 和 rest1_inPlace，然后调用 rest1(rest1_inArray, rest1_inNidm, rest1_inPlace) 函数，将这些参数传递给它。最后，函数将 (elist_array, elist_offset) 作为结果返回。

通过调用 expr() 函数和 rest1() 函数，该函数实现了对表达式列表的处理，并返回相应的结果。在语义分析过程中，这段代码用于解析和处理表达式列表的语法结构。

(3)布尔表达式的翻译

以文法equality为例

```
equility -> rel rest4
```

实现如下

```
def equality():  
    (rest4_inTruelist,rest4_inFalselist)= rel()  
    (equality_truelist,equality_falselist) =  
    rest4(rest4_inTruelist,rest4_inFalselist)  
    return (equality_truelist,equality_falselist)
```

equality() 的函数用于处理等式的语法结构。首先，函数调用 rel() 函数获取关系表达式的真跳转列表和假跳转列表，将结果存储在 (rest4_inTruelist, rest4_inFalselist) 元组中。接下来，函数调用 rest4(rest4_inTruelist, rest4_inFalselist) 函数处理等式的剩余部分，并获取处理后的真跳转列表和假跳转列表，将结果存储在 (equality_truelist, equality_falselist) 元组中。最后，函数返回 (equality_truelist, equality_falselist) 作为结果。

通过调用其他函数，equality() 函数实现了等式的语法结构处理，并返回等式中的真跳转列表和假跳转列表。

(4)控制语句的翻译

```
def backpatch(nextlist,quad):
    for i in nextlist:
        quadruples[i][3] = quad
    return
def merge(l1,l2,l3):
    return l1+l2+l3
```

`backpatch(nextlist, quad)` 函数用于对 `nextlist` 中的每个索引进行遍历，并将对应四元组中的第四个元素（即结果）设为 `quad`。这里假设 `quadruples` 是一个存储四元组的列表，通过 `quadruples[i][3]` 来访问第 `i` 个四元组的第四个元素。该函数的作用是在语义分析过程中，将之前未确定目标的跳转指令的目标位置指定为 `quad`。

`merge(l1, l2, l3)` 函数将三个列表 `l1`、`l2` 和 `l3` 合并成一个新的列表，并将结果返回。该函数在语义分析过程中用于合并不同的列表，用于存储和管理语句的跳转目标或其他需要组合的列表。

以文法 `stmt` 为例

```
stmt -> if(bool) m1 stmt1 n else m2 stmt2 | while(m1 bool) m2 stmt1
```

实现如下

```
def stmt():
    global line_num, col_num
    (token, line_num, col_num) = get_token(line_num, col_num, lines)
    if token == 'if':
        advance()
        advance()
        (bool_truelist, bool_falselist) = bool()
        advance()
        m1_quad = m()
        stmt1_nextlist = stmt()
        n_nextlist = n()
        advance()
        m2_quad = m()
        stmt2_nextlist = stmt()
        backpatch(bool_truelist, m1_quad)
        backpatch(bool_falselist, m2_quad)
        stmt_nextlist = merge(stmt1_nextlist, n_nextlist, stmt2_nextlist)
        return stmt_nextlist
    elif token == 'while':
        advance()
        advance()
        m1_quad = m()
        (bool_truelist, bool_falselist) = bool()
        advance()
        m2_quad = m()
        stmt1_nextlist = stmt()
        backpatch(stmt1_nextlist, m1_quad)
```

```
backpatch(bool_truelist,m2_quad)
stmt_nextlist = bool_falselist
emit('j', '-', '-', m1_quad)
return stmt_nextlist
```

`stmt()` 函数是一个用于处理语句的函数。根据词法单元的值 (`token`) 进行条件判断。

如果 `token` 是 `'if'`，表示当前是一个 `if` 语句。函数会按照顺序执行以下操作：

1. 获取布尔表达式的真假列表。
2. 处理 `if` 语句的第一个分支。
3. 获取 `else` 部分的下一跳位置列表。
4. 处理 `if` 语句的第二个分支。
5. 将布尔表达式为真和为假时的跳转目标设定为对应值。
6. 返回合并后的列表。

如果 `token` 是 `'while'`，表示当前是一个 `while` 循环。函数会按照顺序执行以下操作：

1. 获取布尔表达式的真假列表。
2. 处理 `while` 循环体。
3. 将循环体下一目标设定为第一个标签。
4. 将布尔表达式为真时的跳转目标设定为第二个标签。
5. 返回布尔表达式为假时的跳转目标。

3.测试

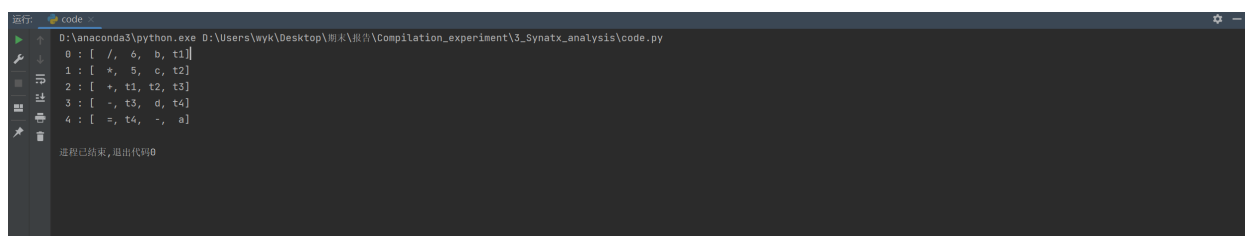
赋值语句的翻译

输入

```
a=6/b+5*c-d;
```

输出

```
0 : [ /, 6, b, t1]
1 : [ *, 5, c, t2]
2 : [ +, t1, t2, t3]
3 : [ -, t3, d, t4]
4 : [ =, t4, -, a]
```



```
运行: Code
D:\anaconda2\python.exe D:\Users\wyk\Desktop\期末\报告\Compilation_experiment\3_Syntax_analysis\code.py
0 : [ /, 6, b, t1]
1 : [ *, 5, c, t2]
2 : [ +, t1, t2, t3]
3 : [ -, t3, d, t4]
4 : [ =, t4, -, a]
进程已结束, 退出代码0
```

数组的翻译

输入

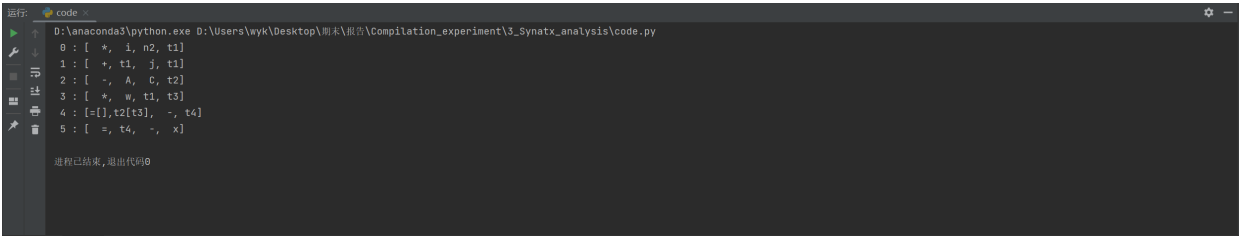
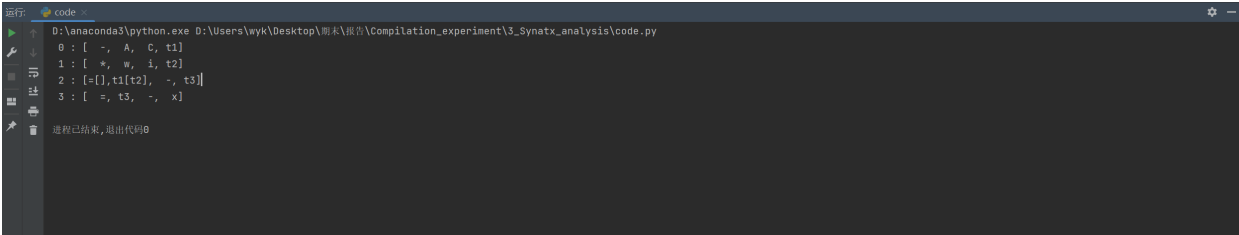
```
x=A[i];

x=A[i,j];
```

输出

```
0 : [ -, A, C, t1]
1 : [ *, w, i, t2]
2 : [=[],t1[t2], -, t3]
3 : [ =, t3, -, x]

0 : [ *, i, n2, t1]
1 : [ +, t1, j, t1]
2 : [ -, A, C, t2]
3 : [ *, w, t1, t3]
4 : [=[],t2[t3], -, t4]
5 : [ =, t4, -, x]
```



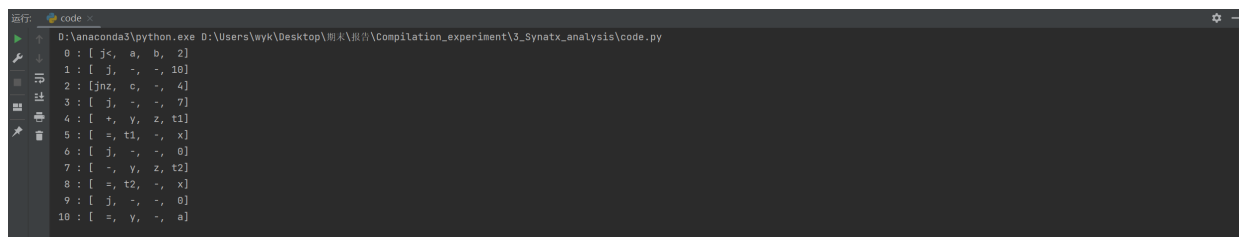
控制语句的翻译

输入

```
while(a<b)
    if(c)
        x=y+z;
    else
        x=y-z;
a=y;
```

输出

```
0 : [ j<, a, b, 2]
1 : [ j, -, -, 10]
2 : [jnz, c, -, 4]
3 : [ j, -, -, 7]
4 : [ +, y, z, t1]
5 : [ =, t1, -, x]
6 : [ j, -, -, 0]
7 : [ -, y, z, t2]
8 : [ =, t2, -, x]
9 : [ j, -, -, 0]
10 : [ =, y, -, a]
```

A screenshot of a code editor window. The title bar says 'code'. The file path is 'D:\anaconda3\python.exe D:\Users\wyk\Desktop\期末\报告\Compilation_experiment\3_Syntax_analysis\code.py'. The code content is the same assembly code as shown in the previous block, with line numbers 0 to 10. The editor has a dark theme and a sidebar on the left with icons for file explorer, search, and other functions.

五、总结

在本次编译原理课程的实践中，我成功实现了编译程序中词法分析器、语法分析器、语义分析和中间代码生成器部分的代码。通过这次实验，我深入了解了编译程序的各个组成部分及其实现原理。

在词法分析器的实现过程中，我学习了如何将输入的源代码字符串转换为一系列的Token序列，并对每个Token进行分类和标记。我了解了正则表达式的使用，以及如何通过有限自动机来实现词法分析器。这个过程让我对源代码的结构有了更清晰的认识。

在语法分析器的实现过程中，我掌握了自顶向下的语法分析方法。通过构建文法规则和递归下降的算法，程序能够根据语法规则生成语法树，并对不符合语法规则的代码进行错误处理。这个过程让我更好地理解了解了语法分析器的工作原理。

在语义分析和中间代码生成器的实现过程中，我了解了如何对语法分析器得到的语法单位赋予相应的语义动作，并计算相应的属性值。我还学习了翻译模式的应用，将源代码转换成四地址代码格式的中间代码。同时，初步了解truelist、falselist和回填技术等知识，对其具体实现方式有了更为清晰的认识。

通过本次课程实践，我对编译原理课程的相关知识有了更加深入的理解。我了解到编译程序是如何将高级语言的源代码转化为可执行代码的关键环节。虽然在未来的工作和科研中，我可能不会直接接触到编译原理相关的内容，但我会将编译原理课程中学习到的相关算法思想运用到后续的学习和科研过程中。总之，本次编译原理实践让我受益匪浅。我不仅掌握了编译程序的基本原理和实现方法，还提高了对代码结构和语法的理解能力，感谢老师在整个实践过程中的指导和帮助！

附录 源代码

```
import os
# 设置相对路径
model_path = os.path.abspath(os.path.join(os.path.dirname(__file__)))
```

```

# 文件读入及初始化
f = open(model_path + '/test.txt', encoding='utf-8')
lines = f.readlines()
line_num = 0
col_num = 0
# Scanner词法分析器部分
def tokenize(code, index):
    operators = ['++', '--', '+', '-', '*', '/', '%', '=', '>', '>=', '<', '<=',
'==', '!=', '&&', '||', '!']
    delimiters = [',', '(', ')', ';', '{', '}', '[', ']']
    keywords = {'main', 'int', 'char', 'if', 'else', 'for', 'while', 'return',
'void'}
    i = index

    while i < len(code):
        if code[i].isspace() or code[i] == '\n':
            return ('', i + 1) # 返回空标记和更新后的索引

        if code[i:i + 2] in operators:
            return (code[i:i + 2], i + 2) # 返回运算符标记和更新后的索引

        if code[i] in operators:
            return (code[i], i + 1) # 返回运算符标记和更新后的索引

        if code[i] in delimiters:
            return (code[i], i + 1) # 返回分隔符标记和更新后的索引

        if code[i].isalpha() or code[i] == '_':
            j = i + 1
            while j < len(code) and (code[j].isalnum() or code[j] == '_'):
                j += 1
            token = code[i:j]
            if token in keywords:
                return (token, j) # 返回关键字标记和更新后的索引
            else:
                return (token, j) # 返回标识符标记和更新后的索引

        if code[i].isdigit():
            j = i + 1
            while j < len(code) and code[j].isdigit():
                j += 1
            return (code[i:j], j) # 返回数字标记和更新后的索引

    return ('', i) # 返回空标记和当前索引

keywords = {'main': 1, 'int': 5, 'char': 3, 'if': 17, 'else': 15, 'for': 6, 'while':
20, 'return': 8, 'void': 9}

symbols = {'<=': 50, '==': 51, '!=': 52, '&&': 53, '||': 54, '!': 55, '++': 56, '--':
57,

```



```

        '+': 41, '-': 42, '*': 43, '/': 44, '%': 45, '=': 46, '>': 47, '>=': 48,
        '<': 49, '(': 81, ')': 82,
        ',': 83, ';': 84, '{': 86, '}': 87, '[': 88, ']': 89}

```

```

def judge_token(s):
    if s[0].isalpha() and s in keywords:
        return keywords[s] # 返回关键字代码
    elif s[0].isalpha() and s not in keywords and s.isalnum():
        return 111 # 返回标识符代码
    elif s.isdigit():
        return 20 # 返回数字代码
    elif s in symbols:
        return symbols[s] # 返回符号代码
    else:
        if len(s) >= 2 and s[0] == '"' and s[-1] == '"':
            return 50 # 返回字符串代码
        else:
            return 'Notfound' # 返回 'Notfound'

def get_token(line_num, col_num, lines):
    if line_num == len(lines):
        return ('%%done', -1, -1)

    (token, col_num) = tokenize(lines[line_num], col_num)

    if token == '':
        if col_num < len(lines[line_num]):
            return get_token(line_num, col_num, lines)
        else:
            return get_token(line_num + 1, 0, lines)
    else:
        if col_num < len(lines[line_num]):
            return (token, line_num, col_num)
        else:
            return (token, line_num + 1, 0)

def advance():
    global line_num, col_num
    (token, line_num, col_num) = get_token(line_num, col_num, lines)

# Parser 语法分析器部分
# Syntx_analysis 语义分析器及中间代码生成部分
quadruples = [] # 四元式列表
temp_count = 1 # 临时变量计数器
nextquad = 0
def emit(op, arg1, arg2, result):
    global nextquad
    quadruple = [op, arg1, arg2, result]

```

```

    quadruples.append(quadruple)
    nextquad += 1
def newtemp():
    global temp_count
    temp_var = "t" + str(temp_count)
    temp_count += 1
    return temp_var
def limit(array,j):
    return 'n'+str(j)
def makelist(i):
    if(i != None):
        return [i]
    else:
        return []
def backpatch(nextlist,quad):
    for i in nextlist:
        quadruples[i][3] = quad
    return
def merge(l1,l2,l3):
    return l1+l2+l3

# 语句和数组
def stmts():
    rest0_inNextlist = stmt()
    stmts_nextlist = rest0(rest0_inNextlist)
    return stmts_nextlist
def rest0(rest0_inNextlist):
    global line_num, col_num
    (token, line_num_, col_num_) = get_token(line_num, col_num, lines)
    if token == 'if' or token == 'while' or judge_token(token) == 111:
        m_quad = m()
        backpatch(rest0_inNextlist,m_quad)
        rest0_inNextlist = stmt()
        rest0_nextlist = rest0(rest0_inNextlist)
        return rest0_nextlist
    else:
        rest0_nextlist = rest0_inNextlist
        return rest0_nextlist
def stmt():
    global line_num, col_num
    (token, line_num_, col_num_) = get_token(line_num, col_num, lines)
    if token == 'if':
        advance()
        advance()
        (bool_truelist,bool_falselist) = bool()
        advance()
        m1_quad = m()
        stmt1_nextlist = stmt()
        n_nextlist = n()
        advance()
        m2_quad = m()
        stmt2_nextlist = stmt()

```

```

        backpatch(bool_truelist, m1_quad)
        backpatch(bool_falselist, m2_quad)
        stmt_nextlist = merge( stmt1_nextlist ,n_nextlist,stmt2_nextlist)
        return stmt_nextlist
    elif token == 'while':
        advance()
        advance()
        m1_quad = m()
        (bool_truelist, bool_falselist) = bool()
        advance()
        m2_quad = m()
        stmt1_nextlist = stmt()
        backpatch(stmt1_nextlist,m1_quad)
        backpatch(bool_truelist,m2_quad)
        stmt_nextlist = bool_falselist
        emit('j','-', '-',m1_quad)
        return stmt_nextlist
    elif judge_token(token) == 111:
        (loc_place,loc_offset) = loc()
        advance()
        expr_place = expr()
        advance()
        if(loc_offset == None):
            emit('=',expr_place, '-',loc_place)
        else:
            emit('[]=', expr_place, '-', loc_place+'['+loc.offset+')')
        stmt_nextlist = makelist(None)
        return stmt_nextlist
def loc():
    global line_num, col_num
    (token, line_num_, col_num_) = get_token(line_num, col_num, lines)
    if judge_token(token) == 111:
        resta_inArray = token
        advance()
        (loc_place,loc_offset) = resta(resta_inArray)
        return (loc_place,loc_offset)
def m():
    m_quad = nextquad;
    return m_quad
def n():
    n_nextlist = makelist(nextquad)
    emit('j','-', '-',0)
    return n_nextlist
def resta(resta_inArray):
    global line_num, col_num
    (token, line_num_, col_num_) = get_token(line_num, col_num, lines)
    if token == '[':
        advance()
        (elist_array,elist_offset) = elist(resta_inArray)
        resta_place = newtemp()
        emit('-',elist_array,'C',resta_place)
        resta_offset = newtemp()

```

```

        emit('*', 'w', elist_offset, resta_offset)
        return (resta_place, resta_offset)
    advance()

    else:
        resta_place = resta_inArray
        resta_offset = None
        return (resta_place, resta_offset)
def elist(elist_inArray):
    expr_place = expr()
    rest1_inArray = elist_inArray
    rest1_inNidm = 1
    rest1_inPlace = expr_place
    (elist_array, elist_offset) = rest1(rest1_inArray, rest1_inNidm, rest1_inPlace)
    return (elist_array, elist_offset)
def rest1(rest1_inArray, rest1_inNidm, rest1_inPlace):
    global line_num, Col_num
    (token, line_num_, Col_num_) = get_token(line_num, Col_num, lines)
    if token == ',':
        advance()
        expr_place = expr()
        t = newtemp()
        m = rest1_inNidm + 1
        emit('*', rest1_inPlace, limit(rest1_inArray, m), t)
        emit('+', t, expr_place, t)
        rest1_inArray = rest1_inArray
        rest1_inNidm = m
        rest1_inPlace = t
        (rest1_array, rest1_offset) = rest1(rest1_inArray, rest1_inNidm, rest1_inPlace)
        return (rest1_array, rest1_offset)

    else:
        rest1_array = rest1_inArray
        rest1_offset = rest1_inPlace
        return (rest1_array, rest1_offset)

# 关系运算
def bool():
    (bool_truelist, bool_falselist) = equality()
    return (bool_truelist, bool_falselist)

def equality():
    (rest4_inTruelist, rest4_inFalselist) = rel()
    (equality_truelist, equality_falselist) =
rest4(rest4_inTruelist, rest4_inFalselist)
    return (equality_truelist, equality_falselist)

def rest4(rest4_inTruelist, rest4_inFalselist):
    global line_num, Col_num
    (token, line_num_, Col_num_) = get_token(line_num, Col_num, lines)
    if token == '==':
        advance()

```

```

        rel()
        rest4()
    elif token == '!=':
        advance()
        rel()
        rest4()
    else:
        (rest4_truelist, rest4_inFalselist) = (rest4_inTruelist, rest4_inFalselist)
        return (rest4_truelist, rest4_inFalselist)

def rel():
    rop_expr_inPlace = expr()
    (rel_truelist, rel_falselist) = rop_expr(rop_expr_inPlace)
    return (rel_truelist, rel_falselist)

def rop_expr(rop_expr_inPlace):
    global line_num, Col_num
    (token, line_num_, Col_num_) = get_token(line_num, Col_num, lines)
    if token == '<':
        advance()
        rop_expr_truelist = makelist(nextquad)
        rop_expr_falselist = makelist(nextquad + 1)
        expr_place = expr()
        emit('j<', rop_expr_inPlace, expr_place, '-');
        emit('j', '-', '-', '-')
    elif token == '<=':
        advance()
        rop_expr_truelist = makelist(nextquad)
        rop_expr_falselist = makelist(nextquad + 1)
        expr_place = expr()
        emit('j<=', rop_expr_inPlace, expr_place, '-');
        emit('j', '-', '-', '-')
    elif token == '>':
        advance()
        rop_expr_truelist = makelist(nextquad)
        rop_expr_falselist = makelist(nextquad + 1)
        expr_place = expr()
        emit('j>', rop_expr_inPlace, expr_place, '-');
        emit('j', '-', '-', '-')
    elif token == '>=':
        advance()
        rop_expr_truelist = makelist(nextquad)
        rop_expr_falselist = makelist(nextquad + 1)
        expr_place = expr()
        emit('j>=', rop_expr_inPlace, expr_place, '-');
        emit('j', '-', '-', '-')
    else:
        rop_expr_truelist = makelist(nextquad)
        rop_expr_falselist = makelist(nextquad + 1)
        emit('jnz', rop_expr_inPlace, '-', '-');
        emit('j', '-', '-', '-')
    return (rop_expr_truelist, rop_expr_falselist)

```

```

#  加减运算
def expr():
    place = term()
    return rest5(place)

def rest5(rest5_in):
    global line_num, col_num
    (token, line_num_, col_num_) = get_token(line_num, col_num, lines)
    if token == '+':
        advance()
        term_place = term()
        rest51_in = newtemp()
        emit('+', rest5_in, term_place, rest51_in)
        return rest5(rest51_in)
    elif token == '-':
        advance()
        term_place = term()
        rest51_in = newtemp()
        emit('-', rest5_in, term_place, rest51_in)
        return rest5(rest51_in)
    else:
        return rest5_in

#  包含乘除的算法表达式
def term():
    place = unary()
    return rest6(place)
def rest6(rest6_in):
    global line_num, col_num
    (token, line_num_, col_num_) = get_token(line_num, col_num, lines)
    if token == '*':
        advance()
        unary_place = unary()
        rest61_in = newtemp()
        emit('*', rest6_in, unary_place, rest61_in)
        return rest6(rest61_in)

    elif token == '/':
        advance()
        unary_place = unary()
        rest61_in = newtemp()
        emit('/', rest6_in, unary_place, rest61_in)
        return rest6(rest61_in)
    else:
        return rest6_in
def unary():
    return factor()
def factor():
    global line_num, col_num

```

```

(token, line_num_, col_num_) = get_token(line_num, col_num, lines)
if token == '(':
    advance()
    expr_ = expr()
    advance()
    return expr_
elif token.isdigit():
    token_ = token
    advance()
    return token_
elif judge_token(token) == 111:
    (loc_place, loc_offset) = loc()
    if (loc_offset == None):
        factor_place = loc_place
    else:
        factor_place = newtemp()
        emit('=[', loc_place+'['+loc_offset+', '-', factor_place)
    return factor_place

else:
    print("error")
    return

# 主函数
stmts()
index = 0
for q in quadruples:
    output = "{:2d} : [{:>3},{:>3},{:>3},{:>3}]" .format(index, q[0], q[1], q[2],
q[3])
    print(output)
    index += 1

```