《编译原理》

实验报告书

班　　级：计　２０３

学　　号：

姓　　名：banban

指导教师：

2023年6月11日

目录

实验一 整数、浮点数混合运算

1、 实验目的

1. 在原有十进制运算基础上添加混合进制的四则运算，二进制 0b 或 0B 开头，八进制 0 开头，十六进制 0x 或 0X 开头。
2. 整型和浮点型混合运算，浮点型包括小数位和指数位。

2、 实验内容和步骤

更改 t_NUMBER(t)函数，使其适配更多进制数字和浮点数的模式，并在函数中对不同进制的数进行转换和返回。

3、 运行结果

```
calc > 2.2
2.2
calc > 2.2+1.1
3.3000000000000003
calc > 0b011*1e2
300.0
calc > 0x02*0b011
6
calc > 0X02+0x01
3
calc > 2+3
5
calc > 0b011*1e2
300.0
```

```
calc > 0x02+0X0a+0001+0b01
14.0
calc > (0x02+0X0a)*0X02
24
calc > 123+0x03*0x04+0b10-07
130.0
calc > 2.2*4
8.8
calc > 1e2+1e2
200.0
calc >
```

4、 心得体会

在原有的简单计算器代码中，我们添加了两个新功能：混合进制运算和整型浮点型混合运算。这些改进使得计算器更加灵活和强大。

要实现混合进制运算，我们需要修改 t_NUMBER(t) 函数，以适应不同进制数字的模式。我们使用正则表达式来匹配二进制、八进制和十六进制数字，并通过 int() 函数将其转换为十进制数。当遇到浮点数时，该函数会将其转换为相应的浮点数格式，并返回该 token。

另外，我们还实现了浮点数的支持，可以在一个表达式中进行整型和浮点型的混合运算。在 t_NUMBER(t) 函数中，我们使用正则表达式识别浮点数的模式，并使用 float() 函数将其转换为浮点数。计算器就能够支持小数位和指数位的浮点数运算，例如 1.2e-3 这样的科学计数法。

5、 关键代码

```
def t_NUMBER(t):
    r'0[bB][01]+|0[oO][0-7]+|0[xX][0-9a-fA-F]+|\d+\.\d*([eE][-+]?\d+)?|\d+([eE][-+]?\d+)?'
    try:
        t.value = int(t.value, 0)
    except ValueError:
        t.value = float(t.value)
    return t
```

实验二 字符串操作

1、 实验目的

在原有功能上增加字符串类型，且字符串支持“+”和“*”功能重载，其中利用“+”实现字符串和数的连接功能时，数可以在“+”的左边，也可以在“+”的右边，且数可以是整型，也可以是浮点型；而“*”实现字符串复制时，需要“*”的左边是字符串，右边是整型数。

2、 实验内容和步骤

1. 更改词法规则，使其能够识别以英文双引号起始，以英文双引号结束，且中间不包括英文双引号的符号串；
2. 更改语法规则，能够实现字符串的识别、字符串的赋值以及字符串的“+”连接和“*”复制功能重载。

3、 运行结果

```
calc > "aa"+"bb"
aabb
calc > "aa"+10
Error: Cannot perform additive arithmetic operations on numbers and strings
None
calc > 10+"aa"
Error: Cannot perform additive arithmetic operations on numbers and strings
None
calc > "aa"-10
Error: Cannot perform subtraction operations on numbers and strings
None
calc > "aa"*2
aaaa
calc > 2*"aa"
aaaa
calc > "aa"*0.1
Error: cannot perform arithmetic on float and string
None
calc >
```

4、 心得体会

在本次我对原有的词法规则和语法规则进行修改，以支持字符串类型和相关功能重载。这个过程中，我深刻感受到编译器设计的核心思想——灵活性和可扩展性。

首先，我们需要更改词法规则，以支持字符串的识别。通过添加相应的正则表达式模式，我们可以轻松地实现对字符串类型的识别，并将其转化为对应的 token。

接着，需要更改语法规则，以支持字符串类型的赋值、连接和复制。这就要求我们掌握语法规则设计的技巧和方法。可以通过定义一个非终结符来表示字符串类型，在相应的产生式中加入对“+”和“*”操作符的支持，从而实现字符串的连接和复制功能。

5、 关键代码

```
tokens = (
    'NAME', 'NUMBER', 'STRING'
)

def t_STRING(t):
    r'\".*?\"'
    t.value = str(t.value[1:-1])  # remove quotation marks
    return t


def t_NUMBER(t):
    r'0[bB][01]+|0[oO][0-7]+|0[xX][0-9a-fA-F]+|\d+\.\d*([eE][-+]?\d+)?|\d+([eE][-+]?\d+)?'
    try:
        t.value = int(t.value, 0)
    except ValueError:
        t.value = float(t.value)
    return t

def p_expression_binop(p):
    '''expression : expression '+' expression
                  | expression '-' expression
                  | expression '*' expression
                  | expression '/' expression'''
    if isinstance(p[1], str) and isinstance(p[3], str) and p[2] == '+':
        p[0] = p[1] + p[3]
    elif isinstance(p[1], str) and isinstance(p[3], float) and p[2] == '+':
        print("Error: Cannot perform additive arithmetic operations on numbers and strings")
    elif isinstance(p[1], str) and isinstance(p[3], int) and p[2] == '+':
        print("Error: Cannot perform additive arithmetic operations on numbers and strings")
    elif isinstance(p[1], int) and isinstance(p[3], str) and p[2] == '+':
        print("Error: Cannot perform additive arithmetic operations on numbers and strings")
    elif isinstance(p[1], float) and isinstance(p[3], str) and p[2] == '+':
```

```
33.         print("Error: Cannot perform additive arithmetic operations on numbers and strings")
34.
35.     elif isinstance(p[1], str) and isinstance(p[3], int) and p[2] == '*':
36.         p[0] = p[1] * p[3]
37.     elif isinstance(p[1], int) and isinstance(p[3], str) and p[2] == '*':
38.         p[0] = p[3] * p[1]
39.     elif isinstance(p[1], int) and isinstance(p[3], int) and p[2] == '*':
40.         p[0] = p[1] * p[3]
41.     elif isinstance(p[1], str) and isinstance(p[3], float) and p[2] == '*':
42.         print("Error: cannot perform arithmetic on float and string")
43.     elif isinstance(p[1], float) and isinstance(p[3], str) and p[2] == '*':
44.         print("Error: cannot perform arithmetic on float and string")
45.
46.     elif (isinstance(p[1], str) or isinstance(p[3], str)) and p[2] == '/':
47.         print("Error: Cannot perform division operations on numbers and strings")
48.     elif (isinstance(p[1], str) or isinstance(p[3], str)) and p[2] == '-':
49.         print("Error: Cannot perform subtraction operations on numbers and strings")
50.
51.     else:
52.         if p[2] == '+':
53.             p[0] = p[1] + p[3]
54.         elif p[2] == '-':
55.             p[0] = p[1] - p[3]
56.         elif p[2] == '*':
57.             p[0] = p[1] * p[3]
58.         elif p[2] == '/':
59.             p[0] = p[1] / p[3]
```

实验三 连续赋值

1、 实验目的

在原有功能上进行更改，使其实现类似 x=y=z=“ABC”，以及 a=b=4 的连续赋值语法规则并打印结果。

2、 实验内容和步骤

1. 更改语法规则，能够实现连续赋值的功能。
2. 对赋值语句添加返回值，使其变量中能够存值。

3、 运行结果

```
calc > a=b=c=111
calc > a
111
calc > b
111
calc > c
111
calc > a+b
222
calc > c=a+b
calc > c
222
calc > a=c+222
calc > a
444
calc > a*2
888
```

```
calc > a=b=c="aaa"
calc > a
aaa
calc > b
aaa
calc > c
aaa
calc > a+b
aaaaaa
calc > c=a+b
calc > c
aaaaaa
calc > c*4
aaaaaaaaaaaaaaaaaaaaaaaa
calc > c*0.1
Error: cannot perform arithmetic on float and string
None
calc > c+100
Error: Cannot perform additive arithmetic operations on numbers and strings
None
calc >
```

4、 心得体会

实现连续赋值是一项非常有用的功能，可以大大提高代码的可读性和简洁性。可以让代码更加清晰明了，避免了过多的重复书写。

在实现连续赋值的过程中，需要对语法规则进行更改，以便识别出连续的赋值操作。通过对输入字符串进行逐个字符的解析，可以判断出哪些部分需要被赋值，然后将其逐个赋值到相应的变量中。这样就能够实现类似 x=y=z=“aaa”，以及 a=b=c=111 的连续赋值语法规则。

为了使得变量中能够存储值，需要对赋值语句添加返回值。这样，在赋值完成后，程序就能够返回相应变量的值，从而方便后续操作。

在编写代码的过程中，需要注意一些细节问题。例如，在解析字符串时，需要考虑空格等特殊字符的影响。同时，在处理连续赋值时，需要保证每个变量都能够正确地获取赋值后的数值。

5、 关键代码

```
def p_statement_assign(p):
    """statement : NAME "=" STRING
                 | NAME "=" statement
                 | NAME "=" expression"""
    names[p[1]] = p[3]
    p[0] = p[3]

def p_statement_expr(p):
    """statement : STRING
                 | expression"""
    print(p[1])

```

实验四 完成指数运算

1、 实验目的

增加语法规则，实现指数运算。

2、 实验内容和步骤

1. 增加终结符号“^”，表示指数运算；
2. 指数的优先级高于取负运算“-”；
3. 指数运算满足右结合。

3、 运行结果

```
calc > 2^2^2
16
calc > 2^1.2
2.2973967099940698
calc > 1.2^2
1.44
calc > 1e2^2
10000.0
calc > (1.2+0.8)^2^2*2
32.0
```

4、 心得体会

在本次语法规则的改进中，增加了终结符号“^”来表示指数运算，并且将指数的优先级设置为高于取负运算“-”，以确保表达式的正确性。实现了指数运算的右结合，使得多个指数运算可以依次执行。

这些改进让我们的语法更加完善，能够支持更多的数学运算式。通过增加“^”符号，可以方便地进行幂运算，例如计算 2 的 3 次方写作“2^3”。同时，考虑到可能会在表达式中使用取负运算，通过设定指数优先级高于取负运算“-”来避免出现错误的计算结果。采用了右结合的方法来处理多个指数运算，比如：2^2^2，这样就可以从右往左依次计算幂次，保证了表达式的正确性和可读性。

5、 关键代码

```
tokens = (
    'NAME', 'NUMBER', 'STRING', 'EXPONENT'
)

t_EXPONENT = r'\^'

precedence = (
    ('right', 'UMINUS'),
    ('left', '+', '-'),
    ('left', '*', '/'),
    ('right', 'EXPONENT'),
)

def p_statement_assign(p):
    '''statement : NAME "=" expression
                 | NAME "=" NAME'''
    global names
    names[p[1]] = p[3]
    if isinstance(p[3], str):
        for key in names.keys():
            if names[key] == p[1]:
                names[key] = p[3]

def p_expression_binop(p):
    '''expression : expression '+' expression
                  | expression '-' expression
                  | expression '*' expression
                  | expression '/' expression
                  | expression EXPONENT expression'''
    if isinstance(p[1], str) and isinstance(p[3], str) and p[2] == '+':
        p[0] = p[1] + p[3]
    elif isinstance(p[1], str) and isinstance(p[3], float) and p[2] == '+':
        print("Error: Cannot perform additive arithmetic operations on numbers and strings")
    elif isinstance(p[1], str) and isinstance(p[3], int) and p[2] == '+':
        print("Error: Cannot perform additive arithmetic operations on numbers and strings")
    elif isinstance(p[1], int) and isinstance(p[3], str) and p[2] == '+':
        print("Error: Cannot perform additive arithmetic operations on numbers and strings")
    elif isinstance(p[1], float) and isinstance(p[3], str) and p[2] == '+':
        print("Error: Cannot perform additive arithmetic operations on numbers and strings")

    elif isinstance(p[1], str) and isinstance(p[3], int) and p[2] == '*':
        p[0] = p[1] * p[3]
```

```
    elif isinstance(p[1], int) and isinstance(p[3], str) and p[2] == '*':
        p[0] = p[3] * p[1]
    elif isinstance(p[1], int) and isinstance(p[3], int) and p[2] == '*':
        p[0] = p[1] * p[3]
    elif isinstance(p[1], str) and isinstance(p[3], float) and p[2] == '*':
        print("Error: cannot perform arithmetic on float and string")
    elif isinstance(p[1], float) and isinstance(p[3], str) and p[2] == '*':
        print("Error: cannot perform arithmetic on float and string")

    elif (isinstance(p[1], str) or isinstance(p[3], str)) and p[2] == '/':
        print("Error: Cannot perform division operations on numbers and strings")
    elif (isinstance(p[1], str) or isinstance(p[3], str)) and p[2] == '-':
        print("Error: Cannot perform subtraction operations on numbers and strings")

    else:
        if p[2] == '+':
            p[0] = p[1] + p[3]
        elif p[2] == '-':
            p[0] = p[1] - p[3]
        elif p[2] == '*':
            p[0] = p[1] * p[3]
        elif p[2] == '/':
            p[0] = p[1] / p[3]
        elif p[2] == '^':
            p[0] = p[1] ** p[3]
```