# 21 Constructing a Compiler Level 3

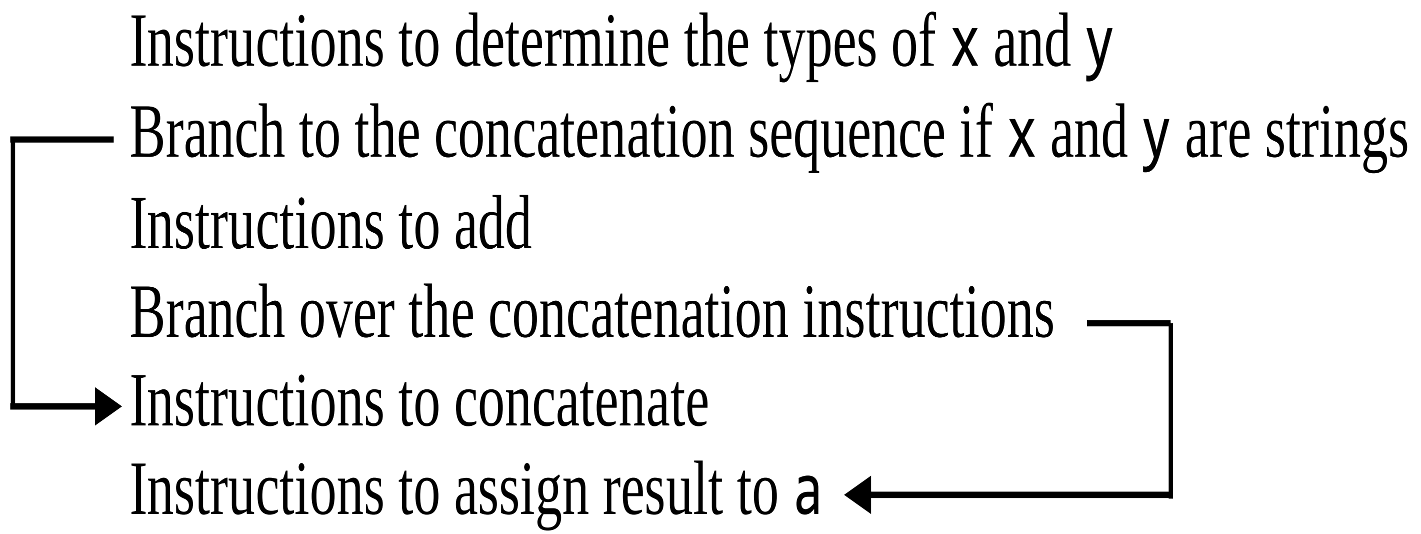
## Introduction

```
s = input('input yes or no: ')
if s == 'yes':
   x = 1             # x and y are integers
   y = 2
else:
   x = 'hello'       # x and y are strings
   y = 'bye'
a = x + y            # addition or concatenation?
```

Source code

Assembler code

```
a = x + y
```



Instructions to determine the types of `x` and `y`
Branch to the concatenation sequence if `x` and `y` are strings
Instructions to add
Branch over the concatenation instructions
Instructions to concatenate
Instructions to assign result to `a`

Figure 21.1

## Representing Dynamically-typed Variables

```
x = 7
y = 'hello'
```

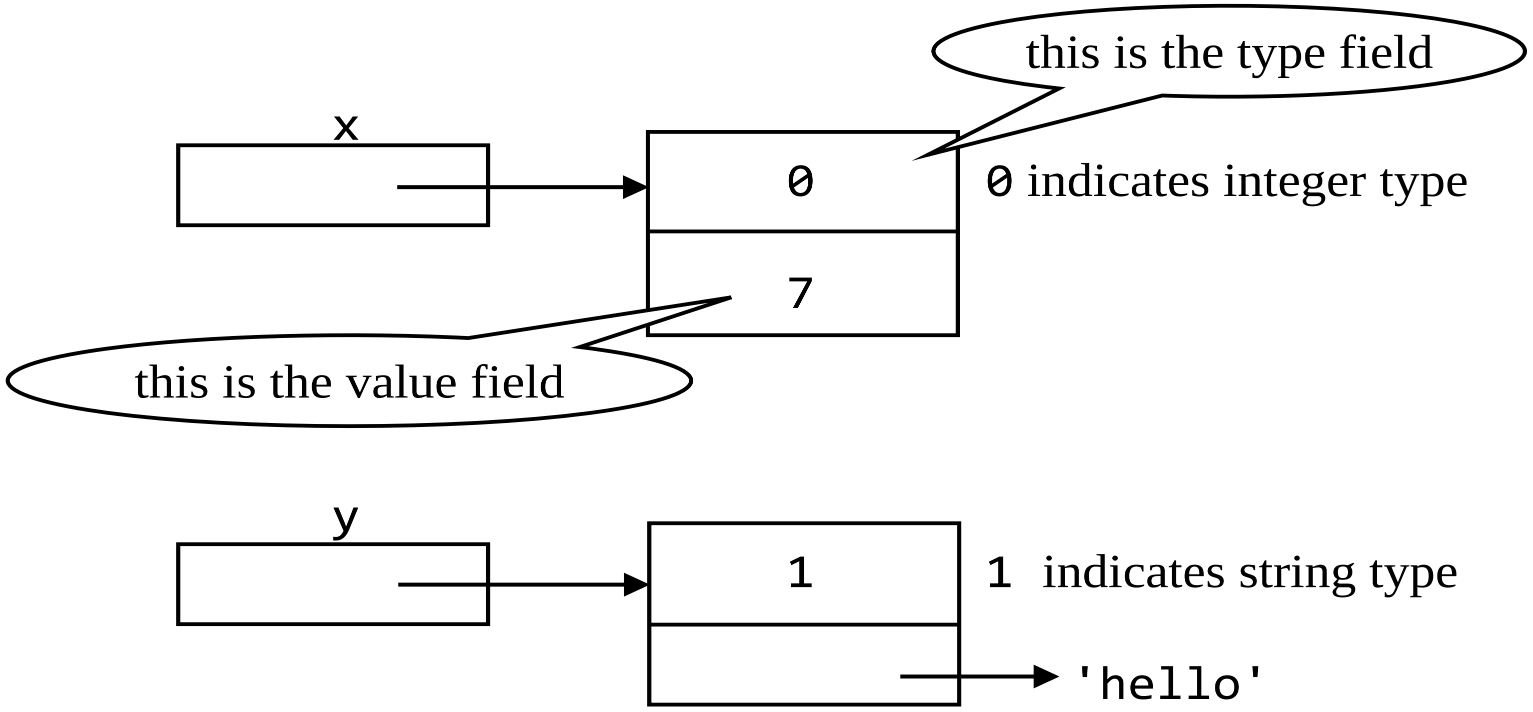


Figure 21.2

```
x = y
```

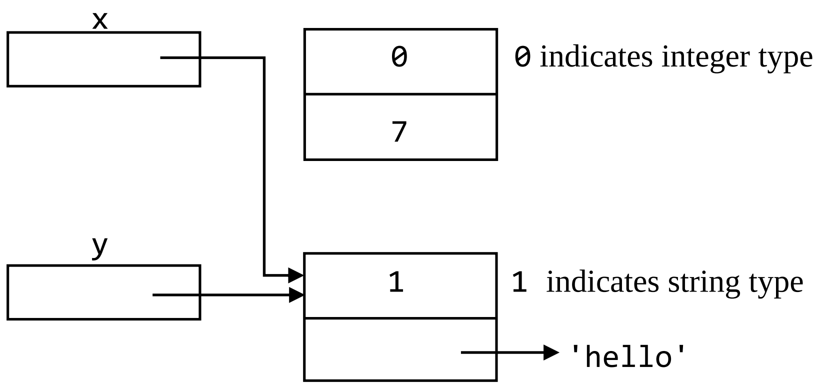

x
y
0
7
0 indicates integer type
1
1 indicates string type
'hello'

Fig. 21.3

```
@ x = y
        ldr r0,  =y          @ get address of y
        ldr r0, [r0]         @ get pointer in y
        ldr r1,  =x          @ get address of x
        str r0, [r1]         @ store pointer in x
```

Figure 21.4

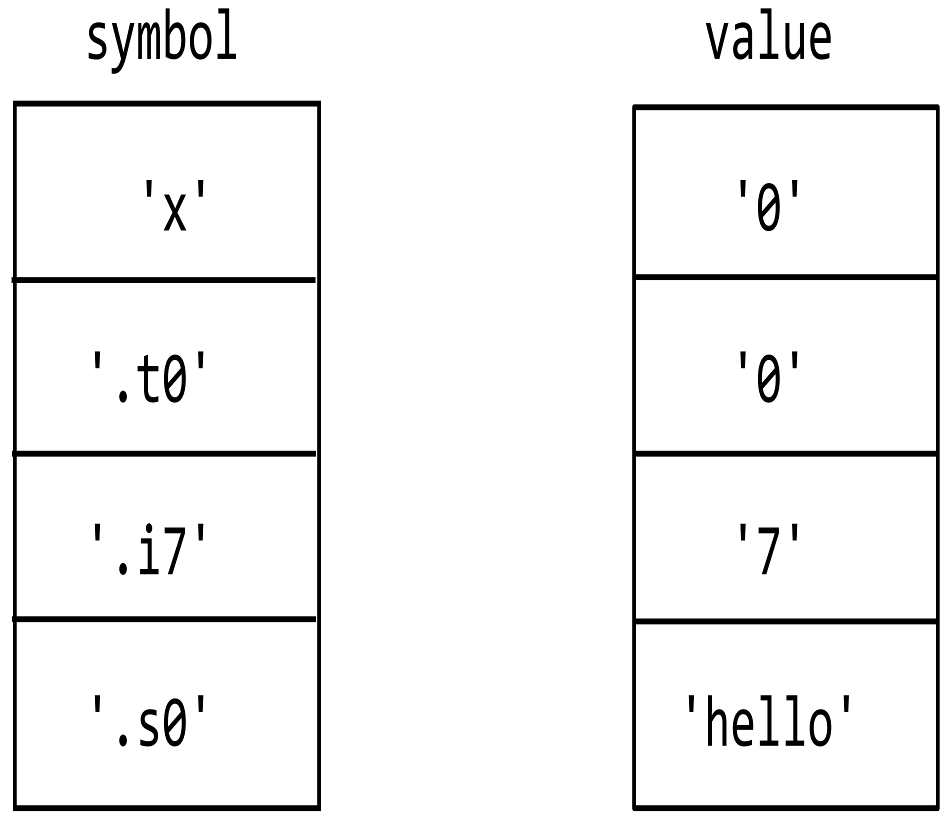


Figure 21.5

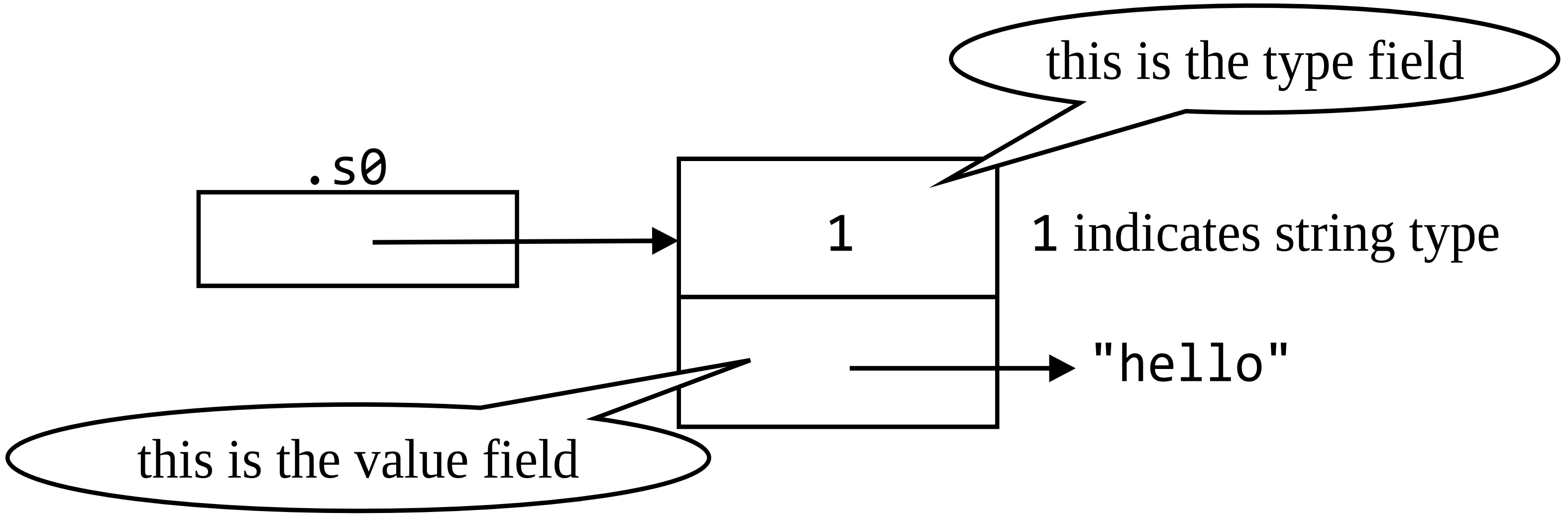
```
if token.lexeme in value:     # check if already in symbol table
   index = value.index(token.lexeme)
else:
   index  = enter('.s' + str(strcount), token.lexeme)
   strcount += 1
advance()
return index
```

```
def parser():
    advance()      # advance to first token
    cg_prolog()    # generates prolog assembler code
    program()      # generates assembler code for program
    cg_epilog()    # generates epilog assembler code
```

```
x:          .word x + 4             @ pointer to object
            .word 0                 @ not assigned anything yet
            .word 0

.t0:        .word .t0 + 4           @ pointer to object
            .word 0                 @ not assigned anything yet
            .word 0

.i7:        .word .i7 + 4           @ pointer to object
            .word 0                 @ 0 indicates integer type
            .word 7                 @ value

.s0:        .word .s0 + 4           @ pointer to object
            .word 1                 @ 1 indicates string type
            .word .s0 + 12          @ pointer to its string
            .asciz "hello"          @ string
```

Figure 21.6

```
.s0:    .word .s0 + 4       @ pointer to object
        .word 1             @ 1 indicates string type
        .word .s0 + 12      @ pointer to its string
        .asciz "hello"      @ string
```



```
    ldr r0, =.s0        @ get address of .s0 into r0
    ldr r0, [r0]        @ load pointer in .s0 into r0 using address in r0


    ldr r1, [r0]        @ get type, loads from address in r0
    ldr r2, [r0, #4]    @ get value, loads from address in r0 plus 4


.i0:      .word .i0 + 4
```

```
size = len(symbol)
i = 0
while i < size:
   if symbol[i].startswith('.s'):    # string?
      outfile.write('%-10s' % (symbol[i] + ':') + '.word ' +
                     symbol[i] + ' + 4\n')
   outfile.write('          .word 1\n')
   outfile.write('          .word ' + symbol[i] + ' + 12\n')
   outfile.write('          .asciz "' + value[i] + '"\n')
else:                       # integer, variable, or temp
   outfile.write(
      '%-10s' % (symbol[i] + ':') + '.word ' + symbol[i] + ' + 4\n')
   outfile.write('          .word 0\n')
   outfile.write('          .word ' + value[i] + '\n')
i += 1
```

Figure 21.7

## Translating Multiplication

```
        ldr r0, =x          @ get address of x
        ldr r0, [r0]        @ get pointer to object
        ldr r1, [r0]        @ get type of object
        cmp r1, #0          @ test if x is an integer
        bne .error          @ branch to .error if not integer
        ldr r2, [r0, #4]    @ get integer value

        ldr r3, =y          @ get address of y
        ldr r3, [r3]        @ get pointer to object
        ldr r4, [r3]        @ get type of object
        cmp r4, #0          @ test if y is an integer
        bne .error          @ branch to .error if not integer
        ldr r5, [r3, #4]    @ get integer value

        mul r0, r2, r5      @ multiply values of x and y

        ldr r1, =.t0        @ get address of temporary variable
        ldr r1, [r1]        @ get pointer to object
        str r4, [r1]        @ store type in object of temp variable
        str r0, [r1, #4]    @ store product in value field
```

Figure 21.8

```
.error:
        mov r0, #1     @ return error code
        pop {pc}       @ pop return address into pc reg
```

## Translating Addition/Concatenation

```
def cg_add(leftindex, rightindex):
   labelstr = cg_getlabel()
   labeltemp = cg_getlabel()
   tempindex = cg_gettemp()
   outfile.write('          ldr r0, =' + symbol[leftindex] + '\n')
   outfile.write('          ldr r0, [r0]\n')       # get ptr to obj
   outfile.write('          ldr r2, [r0]\n')       # r0 has type
   outfile.write('          ldr r1, [r0, #4]\n\n') # r1 has ptr/val
   outfile.write('          ldr r3, =' + symbol[rightindex] + '\n')
   outfile.write('          ldr r3, [r3]\n')       # get ptr to obj
   outfile.write('          ldr r4, [r3]\n')       # r4 has type
   outfile.write('          ldr r5, [r3, #4]\n\n') # r5 has ptr/val
   outfile.write('          cmp r2, r4\n')
   outfile.write('          bne .error\n')
   outfile.write('          cmp r2, #0\n')
   outfile.write('          bne ' + labelstr + '\n')
   outfile.write('          add r0, r1, r5\n')
   outfile.write('          bal ' + labeltemp + '\n')
   outfile.write(labelstr + ':\n')      # concatenate strings
   outfile.write('          ldr r0, =.buf\n')
   outfile.write('          bl strcpy\n')
   outfile.write('          mov r1, r5\n')
   outfile.write('          bl strcat\n')
   outfile.write('          bl strdup\n')
   outfile.write(labeltemp + ':\n')
   outfile.write('          ldr r1, =' + symbol[tempindex] + '\n')
   outfile.write('          ldr r1, [r1]\n') # get ptr to obj
   outfile.write('          str r4, [r1]\n')  # store type
   outfile.write('          str r0, [r1, #4]\n\n') # store value
   return tempindex
```

Figure 21.9

```
def cg_getlabel():
    global labelcount
    label = '.L' + str(labelcount)
    labelcount += 1
    return label
```

Figure 21.10

```
def cg_neg(index):
   outfile.write('          ldr r0, =' + symbol[index] + '\n')
   outfile.write('          ldr r0, [r0]\n')        # get ptr to obj
   outfile.write('          ldr r1, [r0]\n')        # r1 has type
   outfile.write('          cmp r1, #0\n')          # type int?
   outfile.write('          bne .error\n')          # branch if not
   outfile.write('          ldr r2, [r0, #4]\n')    # get value
   outfile.write('          neg r2, r2\n')          # negate value
   tempindex = cg_gettemp()
   outfile.write('          ldr r0, =' + symbol[tempindex] + '\n')
   outfile.write('          ldr r0, [r0]\n')        # get ptr to obj
   outfile.write('          str r1, [r0]\n')        # store type
   outfile.write('          str r2, [r0, #4]\n')    # store value
   return tempindex
```

## Handling Strings in a Compiler

```
     print('hello\nbye')

hello
bye

 hello\nbye



          .asciz "hello
bye"
```

```
@ Thu Feb 22 10:20:43 2018                 YOUR NAME HERE
@ Compiler    = c3.py
@ Input file  = escape.in
@ Output file = escape.s
@------------------------------------------ Assembler code
          .global main
          .text
main:
          push {lr}
```

Depends on what is passed to cg_print()

```
@ print('hello\nbye')
          ldr r0, =.s0            @ get address of arg
          ldr r0, [r0]            @ get pointer to arg’s object
          ldr r2, [r0]            @ get type field
          ldr r1, [r0, #4]        @ get value field into r1 for printf
          cmp r2, #0              @ int or string to be displayed?
          bne .L0                 @ branch if string
          ldr r0, =.fmt0          @ get address of format string for int
          bal .L1
.L0:
          ldr r0, =.fmt1          @ get address of format string for string
.L1:
          bl printf               @ display print statement’s arg
          mov r0, #0              @ start of epilog()-generated code
          pop {pc}
          .data
.fmt0:    .asciz "%d\n"           @ format string for ints
.fmt1:    .asciz "%s\n"           @ format string for strings
.buf:     .space 180              @ buffer for concatenation
.s0:      .word .s0 + 4           @ string constant pointer
          .word 1                 @ type (1 indicates string)
          .word .s0 + 12          @ pointer to string
          .asciz "hello\nbye"
```

\n not replaced

Figure 21.11

This quote is not the terminating quote because it is backslashed.

```
print('A\\')
print('A\\\'')
```

```
elif curchar == "'":  # code in tokenizer for strings
   count = 0
   while True:
      curchar = getchar()
      if curchar == '\n' or curchar == '':
         raise RuntimeError('Unterminated string')
      if curchar == "'" and count == 0: # terminating quote?
         curchar = getchar()      # advance past end of string
         token.category = STRING
         break                    # finished processing string
      if (curchar == '\\'):
         count += 1
         if count == 2:
            count = 0 # reset to 0 on even count
      else:
         count = 0    # reset to 0 if curchar not a backslash
      token.lexeme += curchar
```

Figure 21.12