13 Constructing a Compiler

Changing Gears

```
x = 15
```

h1 hybrid interpreter

```
v = sign*int(token.lexeme)    # convert lexeme to number
```

then enter `v` into `co_consts` if not already there

c1 compiler

`factor` enters '`.i15`' and '`15`' into symbol table

At bottom of program: label created for 15

```
.i15        .word 15
```

Convention for negative constants:

```
.i_15       .word -15
```

Conversion to numbers occurs at assembly time.

Using Both .text and .data Segments

```
ldr r0, =x      @ get address of x from the literal pool
ldr r0, [r0]    @ load x using the address of x in r0
```

Using only a .data segment

```
ldr r0, x
```

x must be range of pc-relative addressing

Minimizing Compiler Complexity

```
    w = x + y

 1          ldr r0, =x       @ get address of x
 2          ldr r0, [r0]     @ get value of x
 3          ldr r1, =y       @ get address of y
 4          ldr r1, [r1]     @ get value of y
 5          add r0, r0, r1   @ add x and y

 6          ldr r1, =.t0     @ get address of .t0
 7          str r0, [r1]     @ store sum of x and y in .t0

 8          ldr r0,  =.t0    @ get address of .t0
 9          ldr r0, [r0]     @ get value of .t0

10          ldr r1, =w       @ get address of w
11          str r0, [r1]     @ store .t0 in w
```

Unnecessary

Figure 13.1

Use a temporary variable to temporarily hold the value of an expression or a subpart of an expression.

Temps: `.t0, .t1`, `.t2`, ...

```
.t0   .word 0
.t1   .word 0
.t2   .word 0
```

1. `a = x + y` `# need to load left and right terms`
2. `a = x*y + z` `# need to load only right term (z)`
3. `a = x + y*z` `# need to load only left term (x)`
4. `a = w*x + y*z` `# no loads needed`

```
w:      .word 0
x:      .word 0
y:      .word 0
.t0:    .word 0
```

```
1 def expr():
2    term()
3    while token.category == PLUS:
4       advance()
5       term()
```

Figure 13.2

```
1 def expr():
2    leftindex  = term()            # get index of left term
3    while token.category == PLUS:
4       advance()
5       rightindex = term()         # get index of right term
6       leftindex  = cg_add(leftindex, rightindex)
7    return leftindex
```

Figure 13.3

Code for `x + y + z`

```
1           @ from first call of cg_add
2           ldr r0, =x          @ get address of x
3           ldr r0, [r0]        @ get x
4           ldr r1, =y          @ get address of y
5           ldr r1, [r1]        @ get y
6           add r0, r0, r1      @ add x and y
7           ldr r1, =.t0        @ get address of .t0
8           str r0, [r1]        @ store sum in .t0
9
10          @ from second call of cg_add
11          ldr r0, =.t0        @ get address of .t0
12          ldr r0, [r0]        @ get .t0
13          ldr r1, =z          @ get address of z
14          ldr r1, [r1]        @ get z
15          add r0, r0, r1      @ add .t0 and z
16          ldr r1, =.t1        @ get address of .t1
17          str r0, [r1]        @ store sum in .t1
```

Symbol Table

```
symbol = []
value = []
```
in string form

```
.i5   .word 5
```
compiler outputs strings

Produced by line 2:

```
          ldr r0, =x
```

```
 1  def cg_add(leftindex, rightindex):
 2     outfile.write('          ldr r0, =' + symbol[leftindex] + '\n')
 3     outfile.write('          ldr r0, [r0]\n')
 4     outfile.write('          ldr r1, =' + symbol[rightindex] + '\n')
 5     outfile.write('          ldr r1, [r1]\n')
 6     outfile.write('          add r0, r0, r1\n')
 7     tempindex = cg_gettemp() # get index of next temp variable
 8     outfile.write('          ldr r1, =' + symbol[tempindex] + '\n')
 9     outfile.write('          str r0, [r1]\n')
10     return tempindex
```

```
1 def enter(s, v):
2     if s in symbol:                # is s already in the table?
3         return symbol.index(s)     # return index of s
4     # otherwise, append s and v and then return index
5     index = len(symbol)            # get index of next slot
6     symbol.append(s)               # append symbol
7     value.append(v)                # append value
8     return index                   # return index
```

1. `term()` calls `cg_mul` instead of `cg_add()`.
2. `term()` initializes the global variable `sign` to 1 before each call of `factor()`. We will explain the purpose of the `sign` variable shortly.

```
 1 def term():
 2     global sign
 3     sign = 1        # initialize sign
 4     leftindex = factor()
 5     while token.category == TIMES:
 6         advance()
 7         sign = 1    # initialize sign
 8         rightindex  = factor()
 9         leftindex = cg_mul(leftindex, rightindex)
10     return leftindex
```

```
1 def cg_gettemp():
2     global tempcount
3     temp = '.t' + str(tempcount) # construct name
4     tempcount += 1               # increment seq number
5     return enter(temp, '0')      # return index of temp
```

```
 1 def factor():
 2    global sign
 3    if token.category == PLUS:
 4       advance()
 5       return factor()
 6    elif token.category == MINUS:
 7       sign = -sign                # change sign for every unary minus
 8       advance()
 9       return factor()
10    elif token.category == UNSIGNEDINT:
11       if sign == 1:    # is number negative or non-negative?
12          index  = enter('.i' + token.lexeme, token.lexeme)
13       else:
14          index  = enter('.i_' + token.lexeme, '-' + token.lexeme)
15       advance()
16       return index
17    elif token.category == NAME:
18       index = enter(token.lexeme, '0')
19       if sign == -1:             # -1 indicate unary minus
20          index = cg_neg(index)   # generate negation code
21       advance()
22       return index
23    elif token.category == LEFTPAREN:
24        advance()
25        savesign = sign     # must save sign because expr()
26        index = expr()      # calls term() which resets sign to 1
27        if savesign == -1:  # so use the saved value of sign
28           index = cg_neg(index)
29        consume(RIGHTPAREN)
30        return index
31    else:
32        raise RuntimeError('Expecting factor')
```

```
1 def cg_neg(index):
2     outfile.write('          ldr r0, =' + symbol[index] + '\n')
3     outfile.write('          ldr r0, [r0]\n')
4     outfile.write('          neg r0, r0\n')
5     tempindex = cg_gettemp()  # tempindex is index of the temp variable
6     outfile.write('          ldr r1, =' + symbol[tempindex] + '\n')
7     outfile.write('          str r0, [r1]\n')
8     return tempindex
```

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

```
def cg_prolog():
   outfile.write('          .global main\n')
   outfile.write('          .text\n')
   outfile.write('main:\n')
   outfile.write('          push {lr}\n')
   outfile.write('\n')
```

```
def cg_epilog():
   outfile.write('\n')
   outfile.write('          mov r0, #0\n')
   outfile.write('          pop {pc}\n')
   outfile.write('          .data\n')
   outfile.write('.fmt0:    .asciz "%d\\n"\n')

   size = len(symbol)
   i = 0
   while i < size:
      outfile.write('%-10s' % (symbol[i] + ':') + '.word ' +
                    value[i] + '\n')
      i += 1
```

assignmentstmt()

```
1 def assignmentstmt():
2    leftindex  = enter(token.lexeme, '0')
3    advance()
4    consume(ASSIGNOP)
5    rightindex = expr()
6    cg_assign(leftindex, rightindex)
```

```
1 def cg_assign(leftindex, rightindex):
2    outfile.write('          ldr r0,  =' + symbol[rightindex] + '\n')
3    outfile.write('          ldr r0, [r0]\n')
4    outfile.write('          ldr r1, =' + symbol[leftindex] + '\n')
5    outfile.write('          str r0, [r1]\n')
```

```
   x = y

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

printstmt()

```
1 def printstmt():
2    advance()
3    consume(LEFTPAREN)
4    index = expr()
5    cg_print(index)
6    consume(RIGHTPAREN)
```

```
1 def cg_print(index):
2    outfile.write('          ldr r0, =.fmt0\n')
3    outfile.write('          ldr r1, =' + symbol[index] + '\n')
4    outfile.write('          ldr r1, [r1]\n')
5    outfile.write('          bl printf\n')
```

```
print(x)
```

```
        ldr r0, =.fmt0        @ get address of format string
        ldr r1, =x            @ get address of x
        ldr r1, [r1]          @ get the value of x into r1
        bl printf             @ call the printf function
```

Commenting the Assembler Code

```
lines = source.splitlines()

lines[token.line - 1].
```

Sample Output File

```
@ Mon Feb 12 13:09:00 2018                    YOUR NAME HERE
@ Compiler    = c1.py
@ Input file  = testcomments.in
@ Output file = testcomments.s
@---------------------------------------- Assembler code
          .global main
          .text
main:
          push {lr}

@ a = 1
          ldr r0,  =.i1
          ldr r0, [r0]
          ldr r1, =a
          str r0, [r1]

@ print(a)
          ldr r0, =.fmt0
          ldr r1, =a
          ldr r1, [r1]
          bl printf

          mov r0, #0
          pop {pc}
          .data
.fmt0:    .asciz "%d\n"
a:        .word 0
.1:       .word 1
```

Information on compile

prolog code

Source code appears as a comment

Source code appears as a comment

epilog code