1 Symbol tables in MIPS

The value of a label is defined to be the number of non-null lines (lines containing an instruction) that precede the label multiplied by 4. To begin, we mark all the non-null lines in the program.

```
begin:
X label: beq $0, $0, after
X jr $4

after:
X sw $31, 16($0)
X lis $4
X abc0: abc1: .word after

loadStore:
X lw $20, 4($0)
X sw $20, 28($0)

end:
```

Now we make a list of labels in the program.

```
begin
label
after
abc0
abc1
loadStore
end
```

Finally, we count the number of non-null lines before each label to get the values.

```
begin 0
label 0
after 8
abc0 16
abc1 16
loadStore 20
end 28
```

This is exactly what the symbol table your program should output in A3P4 will look like. Remember to print it to standard error.
We solved this problem by examining the whole program at once. When writing code that constructs the symbol table, you will have to approach this problem a little differently. The code will probably scan through the program line by line and keep a counter of the number of non-null lines. When the program sees the label, it can compute the label’s value using the counter and then add the label to the symbol table.

2 Bitwise operations in MIPS

Assume that the bitshifts are logical bitshifts, meaning when shifting, vacant digits are always filled with 0s. Note that C++ uses arithmetic shifts, which in particular affects right shifts, depending on if the type is signed or not.

1. \( 3 = 0011 \)
   & \( 5 = 0101 \)
   
   \[ \begin{array}{c}
   \hline
   0001 = 1 \text{ (signed and unsigned)}
   \end{array} \]

2. \( 3 = 0011 \)
   | \( 5 = 0101 \)
   
   \[ \begin{array}{c}
   \hline
   0111 = 7 \text{ (signed and unsigned)}
   \end{array} \]

3. \( 3 = 0011 \)
   \( << 2 \)
   
   \[ \begin{array}{c}
   \hline
   1100 = 12 \text{ (unsigned)}
   = -4 \text{ (signed)}
   \end{array} \]

4. \( 3 = 0011 \)
   \( >> 2 \)
   
   \[ \begin{array}{c}
   \hline
   0000 = 0 \text{ (signed and unsigned)}
   \end{array} \]

5. \( 13 = 1101 \)
   \( << 2 \)
   
   \[ \begin{array}{c}
   \hline
   0100 = 4 \text{ (signed and unsigned)}
   \end{array} \]

6. \( 13 = 1101 \)
   \( >> 2 \)
   
   \[ \begin{array}{c}
   \hline
   0011 = 3 \text{ (signed and unsigned)}
   \end{array} \]

Left shifting allows you to move the pieces of the assembly representation of an instruction to the correct position in the bitpattern that describes the format of the machine code representation of the instruction.

Right shifting allows you to move the bits so that they can be printed by a function which outputs a single byte. Be careful when right shifting in C++; signed and unsigned values give different results when right shifted.

Bitwise Or is the glue that sticks together the various pieces that make up a machine code instruction. Note that when a bit is ‘on’ in either input, that bit will always be on in the output. Thus if you fix one of the inputs, it is possible to turn on bits in the other.

Bitwise And produces a value in which the only bits that are ‘on’ are the ones that were on in both inputs. Thus if you fix one of the inputs, it is possible to turn off bits in the other.
3 Binary output

The code will look like this:

```c
int output_word(unsigned int word) {
    output_byte((word >> 24) & 0xff);
    output_byte((word >> 16) & 0xff);
    output_byte((word >> 8) & 0xff);
    output_byte(word & 0xff);
    return;
}
```

Explanation: Suppose the input word is 0xabcd1234. In binary, this is:

```
1010 1011 1100 1101 0001 0010 0011 0100
a b c d 1 2 3 4
```

We want to output all four bytes of this word from left to right: first 0xab, then 0xcd, then 0x12, then 0x34. But `output_byte` can only output one byte at a time.

We start by figuring out how to output 0xab. We need to manipulate the bits of “word” to get the following binary value:

```
0000 0000 0000 0000 0000 0000 1010 1011
0 0 0 0 0 0 a b
```

This value fits into 8 bits, so we can print it with `output_byte`.

To move the upper 8 bits into the right position, we can use a right bitwise shift by 24 bits, which gives us:

```
1111 1111 1111 1111 1111 1111 1010 1011
f f f f f f a b
```

Why all the 1s? Since 0xabcd1234 begins with a “1” bit, it is a negative two’s complement number. The bitwise shift will preserve the sign, so the number gets padded with 1s instead of 0s. If we had used an `unsigned int` rather than an `int` to store the number being shifted the number would always get padded with 0s.

Now we need to get rid of all the 1s. We can do this by doing a bitwise and with the following number:

```
0000 0000 0000 0000 0000 0000 1111 1111
0 0 0 0 0 0 0 0
```

```
AND 0000 0000 0000 0000 0000 0000 1111 1111
-----------------------------
0000 0000 0000 0000 0000 0000 0000 0000
```

Wherever there is a 0 bit in the bottom number, the result will have a 0. Whenever there is a 1 bit, the result will have a copy of the bit in the top number. So this bitwise and operation will zero out everything but the last 8 bits, which it leaves alone.

After this we can print the byte. So to print the first byte, we do:

```c
output_byte((word >> 24) & 0xff);
```

To print the second and third bytes, we basically do the same thing, but we shift by different amounts. To print the last byte, we don’t need to shift at all; we just need to zero out everything but the last 8 bits. Note that `output_byte` only considers the last 8 bits so zeroing out the other bits is not needed.
To assemble the `.word foo`, we would look up the label “foo” in the symbol table and pass the integer we get (the value of the label) into `output_word`.

```c
output_word(table_lookup("foo"));
```

## 4 C++ Review

### 4.1 Code Style

Here are some problems about the code and possible ways we could improve it:

1. **Inconsistent Spacing.** Indentation and use of spaces should be consistent throughout the program.

2. **Repeated use of long type names.** Mistake can be easily made when typing long, specific sequence of characters over and over again. Furthermore, these code are invulnerable when implementation changes are required. When there are complicated type, we can use typedef.

3. **Redundant if statement.** For example, in foo, the entire function could be replaced by simply returning the result of comparison.

4. **Pass large data structures by value is inefficient, since the entire data structure are copied once the function is called.** Pass by reference are often preferred.

5. **Some code quick be efficiently solved and nicely presented by using the correct data structures.** Make proper use of STL as much as possible. e.g. baz could be done by using std::set.

6. **Magic numbers should be avoided entirely.** All constant value should have a purpose, and a descriptive name.

7. **Use enum constants to create a range of name constants.** e.g. in baz, all string literals could be defined as enum.

8. **Naming for functions and variables should be more descriptive.**

9. **Documentations are always the essential part of good style!**

### 4.2 STL

Try using the STL as much as possible to complete the following exercises:

1. **Write a short C++ program which reads in a sequence of numbers separated by whitespace and prints the sequence twice: once forwards, then once backwards.**

   **Solution:**

   ```c
   #include <iostream>
   #include <algorithm>
   #include <vector>
   #include <iterator>
   #include <string>
   
   using namespace std;

   int main()
   {
     vector<int> seq;
     copy(istream_iterator<int>(cin), istream_iterator<int>(),
         back_inserter(seq));
     copy(seq.begin(), seq.end(), ostream_iterator<int>(cout, "\n"));
     copy(seq.rbegin(), seq.rend(), ostream_iterator<int>(cout, "\n"));
   }
   ```
2. Modify your solution to the previous problem to read in a sequence of name and ID pairs, where the name and ID are separated by whitespace, and each pair is separated from the next pair by whitespace. Print 5 pairs per line, where each pair is formatted as [name,ID]. Avoid using global variables.

Solution:

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
#include <string>
#include <numeric>
using namespace std;

struct Person {
    Person() : id(-1) {} 
    
    string name;
    int id;
};

istream & operator>>(istream & in, Person & p) {
    return in >> p.name >> p.id;
}

ostream & operator<<(ostream & out, const Person & p) {
    return out << ")[" << p.name << "," << p.id << "[";
}

struct Printer {
    ostream & out;
    int count;

    Printer(ostream & out) : out(out), count(0) {} 

    void operator()(Person & p) {
        count = (count+1) % 5;
        out << p << (count == 0 ? "\n" : "\t");
    }
};

int main(){
    vector<Person> seq;
    copy(istream_iterator<Person>(cin), istream_iterator<Person>(),
        back_inserter(seq));
    for_each(seq.begin(), seq.end(), Printer(cout));
    cout << (seq.size() % 5 == 0 ? "" : "\n");
}
```