CS 241 – Week 3 Tutorial

Writing an Assembler Pt. I

Spring 2015

1 Summary

1. Symbol tables
2. Assembly errors
3. C++ Review

2 Problems

1. 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.
```
1  label: label: .word label
2  .word ; 0
3  .word aaaaa
4  .word 1 2 3
5  .word 2147483648 abcde:
6  .word ,
```

Line 1: Duplicate definition of "label".

Line 2: No operand for the `.word`. A `.word` must have exactly one operand.

Line 3: The label "aaaaa" is used here, but it was never defined.

Line 4: Too many operands for the `.word`.

Line 5: Label definitions must appear *before* instructions on a line.

Line 6: The operand must be an identifier (label name), a decimal integer or a hex integer. A comma is none of those.

Note that there is no "out of range" error on line 5. The number here is outside the range of 32-bit two's complement values, but it is within the unsigned range, and the .word operand can be an unsigned *or* two's complement integer.

3 C++ Review

1. Here is a very basic overview of the different kinds of STL Containers that may be useful in this course. There are many reasons why you may choose one container over another. Here are a few

   (a) `pair` is a simple data structure that can hold two values, possibly of differing types. `pair`s are useful for associating values. Keep in mind that when you iterate over the elements in a `map`, each one points to a `pair` type where the `first` field is the key and the `second` field is the value.

   (b) `vector` is a dynamically-sized, array-like abstraction. Consider using a `vector` when you have data that may need to be accessed in a known order.

   (c) `list` is a doubly-linked list. `list` is great for when you need to quickly insert or remove elements from either end.

(d) `map` is an implementation of the dictionary ADT. Use a `map` when you need to associate one piece of data with another.

(e) `set` is a collection of items that provides a fast containment check. Use a `set` when you want to know whether an item has a particular property.

2. Some of the problems with this code are

   (a) Inconsistent spacing. This often arises from mixing tabs with spaces. Different editors may be configured to display tabs differently. For a consistent appearance, most editors can be configured to convert tabs to spaces. Unfortunately, this can cause problems when creating files that require tabs, such as makefiles. The most important thing is to be consistent. If you are going to use tabs, then *always* indent with tabs.

   (b) Repeated use of long type names, e.g. `map<string, map<vector<string>, int> >`. It can be easy to make a mistake when typing long, specific sequences of characters over and over. Furthermore, what if the type needs to change because more information is needed? There may be many places where the type name must change, which can quickly create a lot of busy work. When types begin to get more complicated, it is best to create a custom type with the `class` keyword, or create a type alias using `typedef`.

   (c) Redundant `if` statements. As in `foo`, when checking a condition, if the body of the if statement is simply a boolean, you can often get away with removing the if statement altogether

   (d) Pass large data structures by reference, rather than by value. Copying large, complicated data structures can be inefficient.

   (e) Make proper use of STL data structures. If you find yourself writing code like `baz`, there is a good chance that you could improve it by choosing a suitable data structure.

   (f) Avoid using flag variables. Flag variables quickly complicate your code and obscure meaning. There are some situations where using flag variables can improve code, but they are few and far between. You are better off avoiding them unless you absolutely cannot think of a better way to do something.

   (g) Do not use magic numbers. If you find yourself writing literals such as 143, it would be better to instead create a named constant for that value so that it is clear what the value means.

   (h) Use `enum` constants to create a range of named constants.

   (i) Poor name selection. Avoid names like `foo`, `bar`, `baz`, etc. in favor of evocative names that speak to their purpose.

   (j) Documentation. Comments are for the benefit of readers of your code, including future you. If you decide to return to an uncommented program after years apart, there is a very good chance that you will waste time trying to figure out what's going on, and likely even miss some subtleties that seemed obvious when it was written. It is always better to document assumptions and oddities than it is to assume that your reader will think in the same way that you do.

   One possible improved version of the code is shown below

```
#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <string>
#include <algorithm>
using namespace std;
```

```cpp
typedef map<vector<string>, int> ValueType;
typedef map<string, ValueType> FruitMap;

bool hasManyFruits(const vector<string> &v) {
  return v.size() > 16;
}

int numberMapped(FruitMap &m, const string &f) {
  return m[f].size();
}

bool isFavouriteFruit(const string &fruit) {
  static string init[] = {
    "apple", "pear", "mango", "coconut", "kiwi", "pepper"
  };
  static set<string> favourites(&init[0], &init[sizeof(init)/sizeof(init[0])]);
  return favourites.count(fruit);
}

class OccCheck {
  const string &key;
public:
  OccCheck(const string &key) : key(key) {}

  bool operator()(pair<vector<string>, int> p) {
    int count = 0;
    for (int i = 0; i < p.second; ++i) {
      if (p.first[i] == key) count++;
    }
    return count > p.second/2;
  }
};

enum Mode {
  Apple, Banana, Tangelo
};

const int BadFruitKind = 143;

int main() {
  vector<string> fruits;
  FruitMap fruitMap;
  while (true) {
    string fruit;
    cin >> fruit;

    fruitMap[fruit][fruits] = fruits.size();

    Mode mode;
    if (fruit == "apple") {
      mode = Apple;
    } else if (fruit == "banana") {
      mode = Banana;
    } else if (fruit == "tangelo") {
      mode = Tangelo;
    } else {
      throw BadFruitKind;
    }

    fruits.push_back(fruit);
    if (hasManyFruits(fruits)) {
      cout << "Many␣fruits" << endl;
    }

    int val = numberMapped(fruitMap, fruit);

    if (mode == Banana) {
      break;
    } else if (val > 12345 || fruits.size() > 8 && fruits.size() < 12){
      if (isFavouriteFruit(fruit)) {
        break;
      }
    }
  }

  for (FruitMap::iterator it = fruitMap.begin(); it != fruitMap.end(); ++it) {
    ValueType &v = (*it).second;
    OccCheck occursFrequently((*it).first);
    cout << count_if(v.begin(), v.end(), occursFrequently);
  }
}
```