# Homework Assignment #1

**(due on 04/28 at the beginning of class)**

## Problem #1: History

*(10 point)* Based on your reading of Chapter 2 in the book, answer the following questions:

1. Why was the slowness of interpretation of programs acceptable in the early 1950s?
The bottleneck was software floating point operations
2. Which language is considered the ancestor of COBOL and who designed it?
FLOW-MATIC, designed by Grace Hopper
3. What are two reasons why Java is safer than C++
No pointers, index range checking, JVM, garbage collection
4. What are two languages whose use was mandated by the U.S. Dept. of Defense?
COBOL and Ada
5. Which language first introduced the ability to reference a cross section of an array? (e.g., view the 3rd column of a matrix as a vector)
PL/I
6. What was the main application of the SNOBOL language?
Text processing
7. From which three languages did PL/I borrow many of its constructs?
ALGOL 60, FORTRAN IV, COBOL 60
8. Which language had its syntax described in BNF for the first time?
ALGOL 60
9. Which functional language has a syntax close to that of an imperative language?
ML
10. Which was the first widely used language to do away with punched cards and paper tapes and to rely on an actual terminal?
BASIC

## Problem #2: BNF grammar

*(10 points)* Write the BNF grammar for the Ada `case` statement, as described on page 327 of your book. You can assume that the following nonterminals are defined in the full grammar:

<stmt> a statement
<expr> an expression
<const_expr> a constant expression

<case_stmt> **-->** **case** <expr> **is** <body> **end case;**
<body> **-->** <whenlist> | <whenlist> <whenother>
<whenother> **-->** **when others =>** <stmtlist>
<whenlist> **-->** <whenclause> <whenlist> | <whenclause>
<whenclause> **-->** **when** <const_expr> **=>** <stmtlist>
<stmtlist> **-->** <stmt> ; <stmtlist> | <stmt> ;

## Problem #3: Operator Precedence

Consider the following grammar, **G**:

```
<exp> --> <exp> + <mulexp> | <mulexp>
<mulexp> --> <mulexp> * <divexp> | <divexp>
<divexp> --> <divexp> / <rootexp> | <rootexp>
<rootexp> --> ( <exp> ) | a | b | c | d
```

**a.** *(10 points)* Add to **G** a *right-associative* operator '%', whose precedence is between '+' and '*', thus obtaining a new grammar **G'**.

```
<exp> --> <exp> + <percentexp> | <percentexp>
<percentexp> --> <mulexp> % <percentexp> | <mulexp>
<mulexp> --> <mulexp> * <divexp> | <divexp>
<divexp> --> <divexp> / <rootexp> | <rootexp>
<rootexp> --> ( <exp> ) | a | b | c | d
```

**b.** *(10 points)* Add to **G'** a unary operator '#', whose precedence is the highest, thus obtaining a new grammar **G''**.

```
<exp> --> <exp> + <percentexp> | <percentexp>
<percentexp> --> <mulexp> % <percentexp> | <mulexp>
<mulexp> --> <mulexp> * <divexp> | <divexp>
<divexp> --> <divexp> / <hashexp> | <hashexp>
<hashexp> --> # <hashexp> | < rootexp>
<rootexp> --> ( <exp> ) | a | b | c | d
```

**c.** *(10 points)* Modify **G''** so that the '*' and the '/' operators have the same precendence and are both right-associative.

```
<exp> --> <exp> + <percentexp> | <percentexp>
<percentexp> --> <muldivexp> % <percentexp> | <muldivexp>
<muldivexp> --> <hashexp> <muldiv> <muldivexp> | <hashexp>
<muldiv> --> * | /
<hashexp> --> # <hashexp> | < rootexp>
<rootexp> --> ( <exp> ) | a | b | c | d
```

## Problem #4: Axiomatic Semantics

**a.** *(5 points)* Compute the weakest precondition for the following statement and postcondition (assume that all variables are integers):

```
a = 3 * (b - 1) - 2   { a > 3 }
```

```
{ b > 2 }
```

**b.** *(15 points)* Compute the weakest precondition for the following statement and postcondition (assume that all variables are integers):

```
if (a != 3) a = 2*a - 1; else a -= 2; ;  { a > 0 }
```

By the assignment axiom: `{a > 0} a = 2*a - 1 (S1) { a > 0}`
and by the same axiom: `{a > 2} a -=2 (S2) { a > 0}`

We need to find the weakest `P` such that:

```
{P and (a != 3)} S1 {a > 0}, {P and (a == 3)} S2 {a > 0}
--------------------------------------------------------
         {P} if (a !=3) then S1 else S2 {a > 0}
```

`P = { a > 0 }` is the weakest precondition because:

- `{P and (a !=3)}` implies `{a>0}`, which is the weakest precondition for `S1 {a > 0}`
- `{P and (a == 3)} = {a == 3}`, which is a valid precondition for `S2 {a > 0}`

## Problem #5: Expressions

Consider the following program in C syntax:

```
int foo (int *i) {
   *i *= 3;
   return 1;
}

void main () {
   int x = 5;
   x = x + foo (&x);
   if ((x++) || (foo (&x)))
     x+=10;
   printf("%d\n",x);
}
```

What is the output of this program assuming:

**a.** *(5 points)* Left-to-right operand evaluation, short-circuited boolean expressions

17

**b.** *(5 points)* Left-to-right operand evaluation, non-short-circuited boolean expressions

31

**c.** *(5 points)* Right-to-left operand evaluation, short-circuited boolean expressions

58

**d.** *(5 points)* Right-to-left operand evaluation, non-short-circuited boolean expressions

59

## Problem #6: Storage and Pointers

Consider the following C program (download the `.c` file here):

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define STRINGS 4
char *strings[STRINGS] = {"-one","-two","-three","-four"};

void foo(char *string) {
  static int count=0;
  char *x;
  int i;
  x = (char *)malloc((1 + strlen(string) + strlen(strings[count])) * sizeof(char));
  for (i=0;i<strlen(string);i++) {
    *(x + i) = *(&(string[0]) + i) + 1;
  }
  strcpy(x+strlen(string), *(strings + count));
  *(x+strlen(strings[count])+strlen(string)) = '\0';
  printf("%s\n",x);

  count = (count+1) % STRINGS;
  return;
}

int main(int argc, char **argv) {
  int j;
  for (j=1;j<argc;j++)
     foo(*(argv+j));
}
```

This program goes through all command-line arguments (but for the program name itself), prints out each argument with each ASCII character incremented by one and with "-one" appended to the first argument, "-two" appended to the second argument, "-three" appended to the third argument, "-four" appended to the fourth argument, "-one" appended to the fifth argument, etc. For instance, assume the program is compiled to an executable called `a.out`, the output to the following call would be:

```
    % ./a.out a bb ccc d abc f
    b-one cc-two ddd-three e-four bcd-one g-two
```

(yes, it is a useless program, but it is short and exhibits a variety of storage schemes for variables)

**a.** *(10 points)* For each memory reference below, indicate the area of memory it references (heap area, stack area, static area):

| | |
|---|---|
| `strings` | static |
| `strings[1]` | static |
| `strings[1][2]` | static |
| `string` | stack |
| `string[0]` | static |
| `i` | stack |
| `x` | stack |
| `*x` | heap |
| `count` | static |
| `argc` | stack |

**b.** *(30 points)* Rewrite the program above so that:

1. no program data lives in the data segment;
2. there is no dangling pointer or memory leak when the end of main is reached; and
3. no pointer arithmetic is used (only array references and dereferences).

Yes, it is a pain to write code by hand. We strongly suggest you type it up and print it and turn it in with your homework. And while you're at it, you may want to try to run it :)

```
    #include <stdlib.h>
    #include <string.h>
    #include <stdio.h>
```

```
#define STRINGS 4

void foo(char *string, int count, char **strings) {
   char *x;
   int i;

   x = (char *)malloc((1 + strlen(string) + strlen(strings[count])) * sizeof(char));
   for (i=0;i<strlen(string);i++) {
     x[i] = string[i] + 1;
   }
   strcpy(&(x[strlen(string)]), strings[count]);
   x[strlen(strings[count])+strlen(string)] = '\0';
   printf("%s\n",x);
   free(x);
   return;
}

int main(int argc, char **argv) {
   int j;
   char *strings[STRINGS];
   int count = 0;

   strings[0] = (char *)strdup("-one");
   strings[1] = (char *)strdup("-two");
   strings[2] = (char *)strdup("-three");
   strings[3] = (char *)strdup("-four");

   for (j=1;j<argc;j++) {
     foo(argv[j], count, strings);
     count = (count+1) % STRINGS;
   }

   for (i=0;i<STRINGS;i++)
     free(strings[i]);
   free(strings);
}
```