CSE 413 Final Exam

11 December 2007

Name ___SOLUTION_____ Student ID _____

You should assume that all input data is as specified and is free of errors unless specified otherwise. You may also assume that all calls to library functions (e.g., `malloc`, `free`, etc.) succeed. You do not need to include code to handle errors unless specifically instructed to do so.

Answer all questions; show your work. You may use:

1. Printout of: x86 overview.
2. Printout of: Code generation for D.
3. One 8.5 * 11" piece of paper with handwritten notes

Other items, including laptop computers, calculators, cell phones, and other communications devices, are not allowed.

Advice You have 1 hour and 50 minutes, **do the easy questions first**, and work quickly!

Total: 100 points.

Question	Max Points	Score
1	17	
2	12	
3	10	
4	10	
5	18	
6	18	
7	15	
Total	100	

1. [17 pts.] Consider the following C program (which compiles and runs without errors):

```
#include <stdio.h>

void snowy(int* a, int* b) {
  int c;  c = 37;
  *a = *b;
  printf("In snowy, before rainy: *a=%d, *b=%d, c=%d\n", *a, *b, c);
  *b = rainy(&c);
  printf("In snowy, after rainy: *a=%d, *b=%d, c=%d\n", *a, *b, c);
  }

int rainy(int *x) {
  int b;  b = 50;
  b = b + *x;
  x = &b;
  /* (a) draw picture at this point */
  printf("In rainy: *x=%d, b=%d\n", *x, b);
  return 4;
}

int main(){
  int p;  int q;
  p = 75;
  q = 88;
  snowy(&p, &q);
  printf("In main: p=%d, q = %d\n", p, q);
  return 0;
}
```

(a) Draw a diagram with boxes for each active function showing the situation right when execution reaches the comment in the function `rainy`. Be sure to show the values of all of the variables in each active function. If a variable is a pointer to another variable, indicate its value by drawing an arrow between the variable name and the storage location (variable) that it points to. This is *not* meant to be a question about ebp, esp, stack offsets, etc. This question is very similar to the one we had on the midterm. Draw a picture similar to that one.

(b) What output is produced when this program is executed? If an address is printed out, please make up an address and label the appropriate box in your picture above with that address. (Don't worry about writing out the exact details of each message, but we should be able to tell which values you are printing where.)

In snowy, before rainy: *a=88, *b=88, c=37
In rainy: *x=87, b=87
In snowy, after rainy: *a=88, *b=4, c=37
In main: p=88, q = 4

2. [12] Regular expressions.

(a) Give an English description of the set of strings generated by the regular expression `b(a|b)+a`. (Don't just re-write the rules in English; give a description of the strings, like "all strings of 17 a's followed by 42 b's").

Strings of a's and b's of length greater than 2 (>= 3) that start with b and end with a.

(b) Write a regular expression that generates all strings of 0's and 1's containing an odd number of 1's (a string must contain at least one 1).

0*(0*10*1)*10* OR 0*1(0*10*1)*0*

(c) Write a regular expression that generates all strings containing a's, b's, and c's with at least one a and at least one b.

[abc]*a[abc]*b[abc]* | [abc]*b[abc]*a[abc]*

[abc]*((a[abc]*b) | (b[abc]*a)) [abc]*

3. [10] Consider the context free grammar

S ::= *S T* x | x
T ::= *T* y | y

(a) Give a top down leftmost derivation of the string: x y y x y x

S → STx
-> STxTx
-> xTxTx
-> xTyxTx
-> xyyxTx
-> xyyxyx

(b) Draw a parse tree for the string x y y x y x (the same string as in part (a)).

4. [10] Suppose we want to add the following conditional statement to D:

```
ifequal (exp1, exp2)
  statement1
smaller
  statement2
larger
  statement3
```

The meaning of this is that *statement1* is executed if the integer expressions *exp1* and *exp2* are equal; *statement2* is executed if *exp1* < *exp2*, and *statement3* is executed if *exp1* > *exp2*. Note that `ifequal, smaller,` and `larger` are all keywords.

(a) (5 points) Give context-free grammar production(s) for the `ifequal` statement that allows either or both of the "`smaller`" and "`larger`" parts of the statement to be omitted. If both the "`smaller`" and "`larger`" parts of the statement appear, they should appear in that order. You do not need to give productions for expressions and other types of statements, just the `ifequal` statement (which should be considered a statement as well). Write your grammar here:

Here are two solutions. The first one uses ε-productions

stmt* ::= ifequal (*exp* , *exp*) *stmt* *optsmaller* *optlarger
***optsmaller* ::= smaller *stmt* | ε**
***optlarger* ::= larger *stmt* | ε**

The other one is more brute-force but doesn't include any ε-productions.

stmt* ::= ifequal (*exp* , *exp*) *stmt
| ifequal (*exp* , *exp*) *stmt* smaller *stmt*
| ifequal (*exp* , *exp*) *stmt* larger *stmt*
| ifequal (*exp* , *exp*) *stmt* smaller *stmt* larger *stmt*

(b) (5 points) Is the grammar with your production(s) from part (a) ambiguous? If not, argue informally why not; if it is ambiguous, give an example that shows that it is.

Yes. This grammar has the same sort of problem as the "dangling else" in the usual grammar for conditional statements. There are two possible ways to derive, for example,

ifequal (*exp* , *exp*) ifequal (*exp* , *exp*) *stmt* smaller *stmt*

A derivation can be given where the "smaller" part is associated with the second "ifequal", and another can be given that associates it with the first "ifequal".

5. [18] x86 programming. Consider the following C function.

```
int foo(int x) {
  int temp;
  /* (a)  Show the stack here */
  if (x == 100) {
    temp = x * 2;
  } else {
    temp = bar(x + 5);
  }
  return temp;
}
```

(a) Draw a picture of the stack frame for function `foo` showing the locations of the parameters and local variables plus any other things normally contained in a function stack frame. The picture should show the state of the stack frame at the point right before the first statement in the function is executed (i.e., before any additional items have been pushed on or popped from the stack). Be sure to show the locations referred to by registers `ebp` and `esp`, and the numeric offsets of *all* parameters and local variables

```
foo:
    pushl   %ebp              // prologue
    movl    %esp, %ebp
    subl    $4, %esp          // local vars

    cmpl    $100, 8(%ebp)     // x == 100
    jne     else

    movl    8(%ebp), %eax     // eax <- x
    addl    %eax, %eax        // OR imul $2, %eax
    movl    %eax, -4(%ebp)    // temp <- %eax
    jmp     done

else:
    movl    8(%ebp), %eax     // eax <- x
    addl    $5, %eax          // x + 5

    pushl   %eax
    call    bar
    addl    $4, %esp
    movl    %eax, -4(%ebp)    // temp <- %eax

done:
    movl    -4(%ebp), %eax   // (result is already in eax, so not needed)

    movl    %ebp, %esp        // epilogue
    popl    %ebp
    ret
```

```
foo:
    push   ebp                 // prologue
    mov    ebp, esp
    sub    esp, 4              // local vars

    cmp    [ebp+8], 100        // x == 100
    jne     else

    mov    eax, [ebp + 8]      // eax <- x
    add    eax, eax            // OR imul $2, %eax
    mov    [ebp -4], eax       // temp <- eax
    jmp    done

else:
    mov    eax, [ebp+8]        // eax <- x
    add    eax, 5              // x + 5

    push   eax
    call   bar
    add    esp, 4
    mov    [ebp14], eax        // temp <- eax

done:
    mov    eax, [ebp-4]        // (result is already in eax, so not needed)

    mov    esp, ebp            // epilogue
    pop    ebp
    ret
```

6. [18] Compiler hacking. We would like to extend the D language and compiler to handle a new loop statement of the following form:

```
repeat
  statement1
until ( bool-exp )
```

This is a loop that will always execute once with its test at the bottom. In detail, this new statement executes as follows

1. Execute *statement1*.
2. Evaluate *bool-exp*. If *bool-exp* is true, the loop terminates and execution continues after the loop.
3. If *bool-exp* is false, execution continues by jumping back to the `repeat` keyword, and execution continues at step 1 with *statement1*.

On the next page, write an implementation of compiler function `repeat_until_stmt` that, when called, will parse and generate code for this new kind of loop. You can assume that this function is to be added to a D compiler constructed as described in the compiler assignments. In particular, you should assume the following:

- There is a global variable `current_token` defined in the parser and accessible to all of the functions in the parser. When a parser function is called, this variable contains the first token in the construct that is to be parsed.
- Scanner function `next_token()` can be called at any time to advance `current_token` to the next token in the source program.
- The token header file has new symbols `TOK_REPEAT` and `TOK_UNTIL` defined for the new `repeat` and `until` keywords (if you need to refer to these).
- Functions `statement()`, `exp()`, and so forth are available in the parser to compile each of the major non-terminals in the D grammar.
- A function `new_label(char* lbl)` is available. Each time it is called, it replaces the contents of the string `lbl` with a new, unique string that can be used as a label in the generated assembly language program.
- Function `bool_exp(char* lbl)` parses and compiles a *bool-exp* and generates code that will evaluate the boolean expression and jump to `lbl` if the condition is false.
- Finally, there is a function `gen(...)` that writes its argument as a new line in the assembly language program. Treat this as a pseudo-code function whose argument doesn't need to be strictly legal C as long as the meaning is clear. For example, you can write `gen(jmp lbl);` if you want to generate the instruction "`jmp lbl`" in the program, where `jmp` is a literal opcode and `lbl` is a string variable. (i.e., do this instead of writing detailed string handling code.)
- Use the correct AT&T/Unix syntax in the generated code

6. (cont). Write your compiler function below.

```
/* Parse and compile:
 *   repeat statement until(bool-exp) */
void repeat_until_stmt() {

   char lbl_name[MAX_LABEL_SIZE];
   char lbl[MAX_LABEL_SIZE+1];

   next_token();  // consume repeat

   new_label(lbl_name);
   sprintf(lbl, "%s:", lbl_name);

   gen(lbl);

   statement();

   next_token();  // consume until
   next_token();  // consume (

   bool-exp(lbl_name);

   next_token();  // consume )

   return

}
```

2. [15] Short Answer/Fill in the blank/Multiple Choice:

a) __**Algol 60**___ Programming language designed by committee. Introduced "call by name" semantics.

b) __**Fortran**__ Programming language first developed by John Backus at IBM in 1954. Still in use today.

c) It is easiest to write a recursive descent parser for a language that (pick one):
a) is LL(1), b) has left recursive rules, c) is ambiguous, d) is LR(1)

d) "Functional programming is programming without side effects". In this context, what do we mean by "side effects"?

Side effect = modifying state (assigning values to variables). Can also mean printing things to screen or reading from user.

e) Give 3 examples of things that a compiler might check during the type checking phase (these should not just be the same thing written 3 different ways).

- **Function calls have –**
- **same # of params,**
- **correct type of params,**
- **correct return type**
- **Variables are defined before use**
- **A = b, A and b have compatible types**
- **Only one function of a given name (w. same # of params)**

Extra Credit:

- List one advantage that dynamic linking has over static linking.

  With dynamic linking:
  Executable files are smaller
  Can update libraries easily

- What is a weakness of mark sweep garbage collection: (circle one)
  i. Doesn't collect circular structures
  ii. Big pauses while marking and sweeping
  iii. Extra overhead on every pointer access
  iv. All of the above
  v. None of the above
  vi. i and ii.

- When implementing Object oriented languages using vtables (as described in lecture), how are overridden functions handled? (circle one)
  i. Generate code to search multiple vtables until the function is found
  ii. Wait until runtime to determine which offset to use in a vtable
  iii. Generate code to call the function at a pre-determined offset in a vtable
  iv. All of the above
  v. None of the above
  vi. i and ii.