

Name: _____

USC ID: _____

CS 410 Final Exam Fall 1999 [Bono]

December 14, 1999

There are 13 problems on the exam, with 100 points total available. There are 12 pages to the exam, including this one; make sure you have all of them. If you need additional space to write any answers, you may use the backs of pages (just direct us to look there).

Put your name and USC ID number at the top of the exam. Please read over the whole test before beginning. Good luck!

	value	score
Problem 1	6 pts	
Problem 2	4 pts.	
Problem 3	2 pts.	
Problem 4	2 pts.	
Problem 5	6 pts.	
Problem 6	12 pts.	
Problem 7	4 pts.	
Problem 8	6 pts.	
Problem 9	7 pts.	
Problem 10	13 pts.	
Problem 11	12 pts.	
Problem 12	13 pts.	
Problem 13	13 pts.	
TOTAL	100 pts.	

Problem 1 [6 pts.]

Part A. Thompson's construction is an algorithm to create a

from a _____.

Part B. Subset construction is an algorithm to create a

from a _____.

Part C. _____ is an example of a tool that uses the above algorithms to

automatically create a _____ from a

_____.

Problem 2 [4 pts.]

The frame pointer is a register that points at the current stack frame.

Part A. What program operation could overwrite this value?

Part B. How can we make sure the value doesn't get lost when doing the operation mentioned in part A?

Problem 3 [2 pts.]

Show an example of before-and-after code demonstrating the *reduction in strength* optimization (your example may be given using C++).

Problem 4 [2 pts.]

Apply the *copy propagation* optimization to the following three-address-code sequence. Note: make sure you don't change the meaning of the program. Also, you may write your changes right into the code below.

```
y = x;
```

```
x = y + z;
```

```
y = m * 1;
```

```
p = y * 2;
```

Problem 5 [6 pts.]

Consider the following pseudo-Pascal declaration for a two dimensional array.

```
var myArr : array[2..5, 3..10] of int;
```

Give a formula for computing the offset from the base address of a for the following expression assuming arrays are stored in row-major order and ints are 4 bytes each:

```
myArr[i, j]
```

Problem 6 [12 pts.]

Circle all the items below that could go in a stack frame (a.k.a., activation record):

1. saved register values
2. return address
3. values allocated with "new"
4. global variables
5. temporary variables
6. actual parameters
7. access link
8. the display data structure
9. return value
10. attributes of "self"
11. local variables
12. "static" local variables

Problem 7 [4 pts.]

Part A. What are the two main parsing techniques discussed in this course?

Part B. The Cool parser used which of the above techniques?

Problem 8 [6 pts.]

What is the relationship between context-free grammars, parsers, and context-free languages.
(just one or two sentences will suffice -- there's lots of space because that's the way the formatting came out)

Problem 9 [7 pts.]

Assume we are running an SLR parser. We are in a state that contains the following item:

$A \rightarrow \alpha \cdot$

where A is not the start symbol, and the next input symbol is x

Part A [6]. Circle all of the statements that are true about what we do with the stack on this step of parsing. (Note: this only concerns changes related to grammar symbols, not states.)

1. push x if it is in FOLLOW(A)
2. push A if x is in FOLLOW(A)
3. accept if $x = \$$
4. push x if it is in FIRST(A)
5. push x if it is in FIRST(α)
6. push A if x is in FOLLOW(B) where α has the form $\alpha' B$
7. pop top symbol from the stack
8. pop the handle from the stack

Part B [1]. Circle the statement that is true about the *state* to push for this parse step.

1. the state to push is part of the action table entry
2. the state to push is given by an entry in the goto table
3. push no state because we accepted

Problem 10 [13 pts.]

Consider the following basic block. Assume all of the programmer variables, a through d, are live at the end of the basic block, and the temporary variables, t1 through t6, are not live at the end of the basic block.

```
t1 = a * b;
t2 = c - d;
t3 = t1 + t2;
a = t3;
t4 = a * b;
t5 = c - d;
t6 = t4 + t5;
b = t4;
```

Part A. Show the DAG representation for this basic block.

Part B. Generate an improved 3AC sequence from your DAG.

Problem 11 [12 pts.]

Write a routine, `cgen`, to generate code for a `while` loop. More details below:

- about the `while` loop: the `while` loop is like the one in C++, in that you can give an arbitrary integer-valued expression for the condition, and a non-zero value is interpreted as true, and zero is interpreted as false.
- the code generation scheme will be similar to the one we discussed in lecture for a stack-machine:
 - temporaries are pushed on the stack
 - the results of expressions are always left in `$a0`
 - evaluating an expression leaves the stack pointer, `$sp`, with the same value it had before evaluating the expression
 - all results will be 32-bit integers (NOTE: this is different than operations in `cool`).
 - we'll show the input of the routine to be the `while` loop itself; you can call `cgen` recursively on parts of the expression to generate code for those parts
- you may use the function `newLabel()` to generate unique labels (returns a `char*`).
- output your code by using `<<` statements to `cout`.
- syntax of the assembly language to generate (we'll call it pseudo-MIPS). It has operations like MIPS, but allows you to take operands directly from memory:

```
x := y           For loading, storing, and moving between registers
x := y op z      Where op is one of +, -, *, /, %, "and", "or"
x := op y        Where op is one of -, "not"
goto L           jump to label L
if y relop z goto L    Conditional jump to label L, where relop
                    is one of <, >, =, !=, >=, <=
push y           shorthand for: 0($sp) := y; $sp := $sp-4;
pop              shorthand for: $sp := $sp + 4;
```

in the above statements:

```
"x", "y", and "z" may be one of:
    a register
    offset(reg)
```

and "y" and/or "z" may also be constant values

(Continued on next page)

Problem 11 (cont.)

cgen(**while** (**condn**) **body**) =

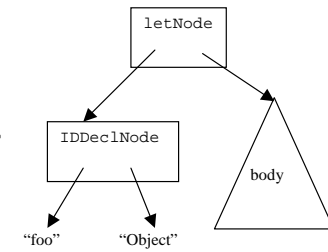
Problem 12 [13 pts.]

Recall that the Cool *let* expression labeled (i) below has the same meaning as the expression given below it, labeled (ii). In fact, we built the same AST for the two expressions.

```
let a : T1, b : T2, c : T3 in      // (i)
  body
end
```

```
let a : T1 in                    // (ii)
  let b : T2 in
    let c : T3 in
      body
    end
  end
end
```

On the right is a diagram of an example AST that gets built for a let with *one* identifier:



Part A [3]. Show the AST that gets built for the single let with three identifiers, labeled (i), given near the top of the page.

Problem 12 (cont.)

Part B [10]. Give a grammar for `LetExpr` along with semantic rules to create the correct AST. I.e., parsing with this grammar and semantic rules the parser can recognize a let with multiple identifiers while building the tree as if it were nested lets of single identifiers. Some more details:

- your grammar may use the nonterminal `Expr` that you don't have to define (Note: one of the forms `Expr` can take is `LetExpr`).
- the body of the let is an `Expr`
- for the purposes of this problem, you may assume a simpler version of let that has no optional initialization expression.
- you can use the tokens `ID` and `TYPE` for the identifier and type, respectively.
- you can use the attribute `ID.value` and `TYPE.value` to access the lexemes associated with each of these tokens (they are type `Symbol`).
- your semantic rules should compute the attribute `LetExpr.tree`, which will be the AST attribute used for all expressions.
- Assume we have the following functions for building pieces of the let AST which you may use in your semantic rules:

```
ExprTree makeLet(IdDeclTree declPart, ExprTree body);
```

```
IdDeclTree makeIdDecl(Symbol id, Symbol type);
```

`LetExpr` →

Problem 13 [13 pts.]

Given the following program in a language allowing nested procedures, show the pointers for getting to non-local references at the following snapshots of the program running, shown by what's on the call stack at that point. For each snapshot give the answer twice: once, assuming you're using a display, and another time assuming you are using access links.

```
void main() {  
    void bar() {  
        . . . // body of bar  
    }  
    void blob() {  
        void baz() {  
            . . . // body of baz  
        }  
        . . . // body of blob  
    }  
    . . . // body of main  
}
```

Part A.

Access links

main
bar
blob
baz
blob

Display

main
bar
blob
baz
blob

Part B.

Access links

main
blob
blob
baz

Display

main
blob
blob
baz