

Name: Fall 2003 Final Exam Solution

USC loginid: SOLUTION

CS 410 Final Exam
Fall 2003 [Bono]
December 10, 2003

There are 13 problems on the exam, with 92 points total available. There are 10 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 loginid at the top of the exam. Please read over the whole test before beginning. Good luck!

	value	score
Probs 1 - 3	10 pts.	
Problem 4 - 5	6 pts.	
Problem 6	5 pts.	
Problem 7	4 pts.	
Problem 8	15 pts.	
Problem 9 - 10	9 pts.	
Problem 11	9 pts.	
Problem 12	15 pts.	
Problem 13	19 pts.	
TOTAL	92 pts.	

Problem 1 [4 pts.]

Part A [2]. A symbol table for one scope is a collection of key-value pairs. During semantic analysis, what is the value we associate with a key?

its type

Part B [2]. Symbol tables built during semantic analysis are still useful in the code generation phase. Name a value a compiler might add to existing symbol table entries during code generation (i.e., an additional value to associate with the key).

Several Possible Answers: its offset; register assignments; liveness next use; its address

Problem 2 [4 pts.]

Short answer problems about semantic analysis. For the following questions assume have an AST built during the parsing phase, and that we have already built the inheritance hierarchy.

Part A [1]. Processing the *attribute and method declarations* in Cool can be done in one pass over the classes by processing the *classes* in what order?

- basic classes followed by user-defined classes
- preorder traversal of inheritance hierarchy**
- postorder traversal of inheritance hierarchy
- any order works

Part B [1]. Processing the *expressions* in Cool can be done in one pass over the classes by processing the *classes* in what order?

- basic classes followed by user-defined classes
- preorder traversal of inheritance hierarchy
- postorder traversal of inheritance hierarchy
- any order works**

Part C [2]. The two tasks mentioned in parts A and B must each be done as a separate pass. What scoping rule(s) of Cool necessitates the two passes? (Note: this is *not* a rule in a lot of other languages, including C++.)

You can use some names before they are defined (eg attributes, methods, classes)

Problem 3 [2 pts.]

The parser *bison* constructs is most similar to which kind of parser listed below:

- SLR(1)**
- LL(1)
- DFA
- recursive descent

Problem 4 [2 pts.]

What formal language is used to describe patterns for tokens for lexical analysis:

regular expressions

Problem 5 [4 pts.]

Cool objects have information about what class they are stored right in the object at runtime. Name two features of Cool that requires type-specific information for an object at run-time:

1. dynamic dispatch
2. case expressions

Problem 6 [5 pts.]

In class we discussed that a useful application of *dead-code elimination* would be for code that is only used in the debugging phase of program development. In PA4 much of our output was controlled by a debug flag `semant_debug`. E.g., `semant.cc` has code such as,

```
if (semant_debug) {  
    <code to print our inheritance hierarchy>  
}
```

This flag gets turned on when we run with the `-s` flag. E.g.,

```
mysemant -s testprog.cl
```

Can the C++ compiler eliminate the debug code from our `semant.cc`? **YES / NO**
Why or why not?

Answer: NO. The value of `semant_debug` is set at run time (e.g. one run with “-s”, one run without)

Problem 7 [4 pts.]

Would it be possible to apply the *loop version of reduction-in-strength* optimization to the following loop? **YES / NO** Why or why not?,

```
for (int i = 0; i < n; i+=2) {  
    A[i] = readData(cin);  
}
```

Answer: YES. Offset for array changes by the same fixed amount every iteration, so you can replace the multiply (in `A[i]`) to compute the offset by an addition.

(e.g. if array holds ints, add 8 every time, instead of `i*2*4`)

Problem 8 [15 pts.]

Consider the following grammar over the alphabet $\{(,)\}$:

$$\begin{aligned} S &\rightarrow S S \\ &| (S) \\ &| () \end{aligned}$$

Part A. Do any grammar transformations necessary to make the grammar suitable for LL(1) parsing. Circle and label each of the resulting grammar(s) with the transformation(s) you applied (or say “same” for that part if no transformations were necessary).

Part B. Attempt to create the LL(1) parse table for the final grammar you gave in part A. Show the complete table: if this grammar is not LL(1), that is if there are conflicts, show all the values that could go in a table entry.

Show your work.

Circle and label your answers to each of the parts.

Answer:

PART A

1. Eliminate Left Recursion

$$\begin{aligned} S &\rightarrow (S) T \\ &| () T \\ T &\rightarrow S T \\ &| \epsilon \end{aligned}$$



2. Left Factor

$$\begin{aligned} S &\rightarrow (R \\ R &\rightarrow S) T \\ &|) T \\ T &\rightarrow S T \\ &| \epsilon \end{aligned}$$

PART B

$$\text{FIR}(S) = ($$

$$\text{FIR}(R) = (,)$$

$$\text{FIR}(T) = (, \epsilon$$

$$\begin{aligned} \text{FOL}(S) &= \$,) \cup \text{FIR}(T) - \{ \epsilon \} \cup \text{FOL}(T) \\ &= \$,), (\end{aligned}$$

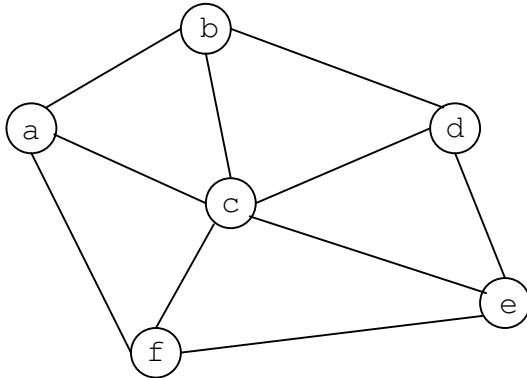
$$\text{FOL}(T) = \text{FOL}(R) = \text{FOL}(S) = \$,), ($$

	()	\$
S	(R		
R	S) T) T	
T	ST / ϵ	ϵ	ϵ

CONFLICT

Problem 9 [4 pts.]

Consider the following *register interference graph* for variables a-f.



Part A. Name two variables whose values would have to be stored in *different* registers.

Answer: **a and b** [Any two connected by an edge]

Part B. Name two variables whose values could be stored in the *same* register, or write *none* if there are none.

Answer: **a and e** [Any two *not* connected by an edge]

Problem 10 [5 pts.]

Give a formula for computing the byte offset for the array reference $\mathbf{a[i][j][k]}$, given the C array declaration shown below. Assume that arrays are stored in row-major order, and that `ints` are 4 bytes long. Hint: in row-major order, the rightmost subscript changes fastest when looking at where n-dim indices are stored in memory (e.g., the first three words of the array contain $\mathbf{a[0][0][0]}$, $\mathbf{a[0][0][1]}$, $\mathbf{a[0][0][2]}$, etc.).

```
int a[DIM1][NROWS][NCOLS];
```

Answer: $4 * (i * NROWS * NCOLS + j * NCOLS + k)$

Alternate Solution : $4*(NCOLS*(NROWS*i+j)+k)$

Problem 11 [9 pts.]

Show the values printed by the following program assuming various parameter-passing schemes given below:

```
int A[2];

void swap (int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
    A[1] = 2;
}

main ()
{
    int i, j;
    i = 1;
    A[0] = 4;
    A[1] = 3;
    swap (i, A[i]);
    cout << i << " " << A[0] << " " << A[1] << endl;
}
```

Part A. Call by value

Answer: 1 4 2

Part B. Call by reference

Answer: 3 4 2

Part C. Call-by-value-return

Answer: 3 4 1

Problem 12 [15 pts.]

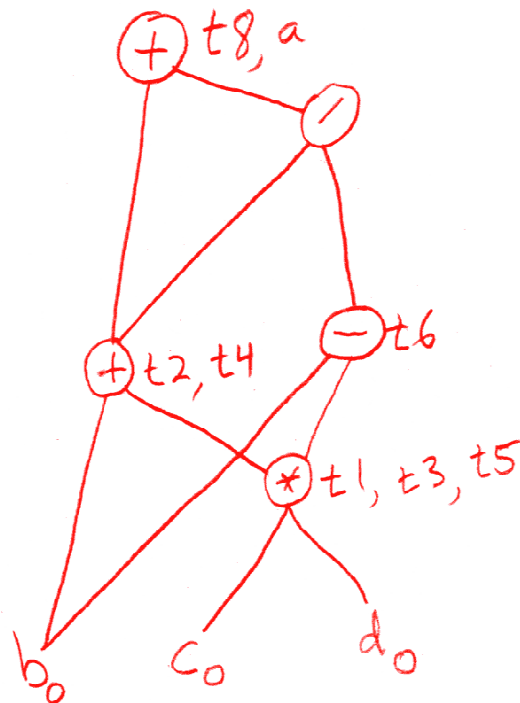
Consider the following basic block. Assume all of the programmer variables, a through d , are live at the end of the basic block, and the temporary variables, t_1 through t_8 , are not live at the end of the basic block. *Label and circle your final answers for each part.*

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

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

```
t1 = c * d;  
t2 = b + t1;  
t3 = c * d;  
t4 = b + t3;  
t5 = c * d;  
t6 = b - t5;  
t7 = t4 / t6;  
t8 = t2 + t7;  
a = t8;
```

Answer: PART A



Answer: PART B

```
t1 := c * d;  
t2 := b + t1;  
t6 := b - t1;  
t7 := t2 / t6;  
a := t2 + t7;
```

Problem 13 [19 pts.]

Consider a new C statement, **if3**. **if3** is a three-way if that goes to a different branch depending on the whether a comparison comes out as **<**, **==**, or **>**. Here is its syntax:

```
stmt → if3 (expr1 ? expr2) {  
    < : stmt ;  
    == : stmt ;  
    > : stmt ;  
}
```

This new statement would be useful for implementing binary search. For example:

```
while (!emptyRange(low, high) && (!found)) {  
    mid = (low + high) / 2;  
    if3 (targetValue ? arr[mid]) {  
        < : high = mid - 1;  
        == : found = true;  
        > : low = mid + 1;  
    }  
}
```

Part A [4]. If we added this construct to a C compiler, would the lexical analyzer need to be changed? **YES** / **NO**

If so how? (i.e., describe the rules that would be needed to be added to a flex file). If not, why not?

Answer: YES. Add pattern if3

Part B [15]. Write a C++ member function for generating 3-address code for this new construct from an AST representation of the code. Here is the AST structure for this construct:

```
class if3 : Statement {  
protected:  
    Expression *expr1;  
    Expression *expr2;  
    Statement *LTstmt; // less than (LT) branch  
    Statement *EQstmt; // equal to (EQ) branch  
    Statement *GTstmt; // greater than (GT) branch  
public:  
    virtual void genCode(); // you write this function  
    . . .  
};
```

(Don't put your answer here. Problem continued next page.)

Problem 13 (cont.)

Assumptions about generating code:

- `newTemp()` generates unique temporary names (returns a string).
- `newLabel()` generates unique labels (returns a string).
- There is a polymorphic `genCode()` function defined for all AST node types.
- All `Expression` node types have a `string` field called `place` for storing the name that will hold its value. `genCode` on the expression initializes this field. `place` can be accessed via the member function `getPlace()`.
- you may use the function `emit`, which takes a variable number of arguments, to generate 3AC statements. Here is an example of its use, where the parts in double-quotes are literal, and `myTemp` is a string variable:

```
emit(myTemp, ":", myTemp, "+27;");
```

- Syntax of the 3AC for the generated code:

`x := y` Copy `y`'s value into `x`

`x := y op z` Where `op` is one of `+`, `-`, `*`, `/`, `%`, `or`, and

`x := op y` Where `op` is one of `-`, `not`

`goto L` Jump to label `L`

`x := y[i]` Set `x` to the value in the mem. loc. `i` bytes beyond `y`

`x[i] := y` Give the mem loc `i` bytes beyond `x` the value in `y`

`if y relop z goto L` Conditional jump to label `L`, where `relop` is one of `<`, `>`, `=`, `!=`, `>=`, `<=`

where the operands can be names or integer constants (except those used as arrays), and the result must be a name. For logical operations, assume non-zero is true, and zero is false.

(Don't put your answer here. Problem continued next page.)

Note: A good strategy would be to write out an outline of the logic of what the generated code would look like. Like this:

[E1.code]

[E2.code]

if (e1.val >= e2.val) goto GEbranch

[LTstmt.code]

goto end

GEbranch: if (e1.val > e2.val) goto GTbranch

[EQstmt.code]

goto end

GTbranch: [GTstmt.code]

end:

Problem 13 (cont.)

Tree structure reprinted for your convenience:

```
class if3 : Statement {
protected:
    Expression *expr1;
    Expression *expr2;
    Statement *LTstmt; // less than (LT) branch
    Statement *EQstmt; // equal to (EQ) branch
    Statement *GTstmt; // greater than (GT) branch
public:
    virtual void genCode(); // you write this function
    . . .
};
```

```
void if3::genCode()
{
```

Answer:

```
    string GEbranch=newLabel();
    string GTbranch=newLabel();
    string end=newLabel();
    e1->genCode();
    e2->genCode();
    emit("if ", e1 . getPlace(), ">= ",
        e2->getPlace(), "goto ", GEbranch);
    LTstmt->genCode();
    emit("goto ", end);
    emit(GEbranch, ":");
    emit("if ", e1->getPlace(), ">",
        e2->getPlace(),
    "goto ", GTbranch);
    EQstmt->genCode();
    emit("goto ", end);
    emit(GTbranch, ":");
    GTstmt->genCode();
    emit(end, ":");
}
```