

# Final Exam

## CS 410, Fall 1997

December 11, 1997

There are 10 problems on the exam, with 100 points total available. There are 10 pages to the exam, including this one, make sure you have all of them. Don't forget to put your name at the top of the exam. Please read over the whole test before beginning. Good luck!!!

We use the following conventions for grammars on the exam, except where noted:

- epsilon, the empty string is represented by  $\epsilon$
- terminals are represented by lower-case letters and punctuation symbols
- non-terminals are represented by upper-case letters

	value	grade
Problem 1	9 pts.	
Problem 2	6 pts.	
Problem 3	13 pts.	
Problem 4	8 pts.	
Problem 5	14 pts.	
Problem 6	12 pts.	
Problem 7	6 pts.	
Problem 8	5 pts.	
Problem 9	10 pts.	
Problem 10	17 pts.	
TOTAL:	100 pts.	

## **Problem 1 [9 pts]**

Short answer:

**Part A. [1 pts]** Regular expressions are useful for creating what part of a compiler?

**Part B. [1 pts]** An abstract machine for recognizing a regular set is called a

**Part C. [3 pts]** Bison is a tool for automatically generating a  
from a

**Part D. [4 pts]** A LR parser acting on an input string corresponds to what type of derivation?

## **Problem 2 [6 pts]**

Give a definition of and an example of an ambiguous grammar.

### Problem 3 [13 pts]

**Part A [10 pts]** Eliminate left recursion from the following grammar:

```
S  ->   S a b
      |   S a S
      |   X
```

```
X  ->   X c
      |   a
      |   b
```

**Part B [3 pts]** Eliminating left recursion is necessary for what parsing technique?

### Problem 4 [8 pts]

What does the following program print under each of the following assumptions:

- lexical scoping and pass-by-value
- lexical scoping and pass-by-reference
- dynamic scoping and pass-by-value
- dynamic scoping and pass-by-reference

```
int x = 5;

void f(int y) {
    y = y * x;
    x = x - y;
}

main()
{
    int x = 2;

    f(x);

    cout << x;
}
```

## Problem 5 [14 pts]

Consider the following augmented grammar:

- (0)  $S' \rightarrow S$
- (1)  $S \rightarrow X S X$
- (2)  $S \rightarrow c$
- (3)  $X \rightarrow a$
- (4)  $X \rightarrow b$

Build the SLR parsing tables for this grammar. Here is the canonical collection of sets of LR(0) items to get you started:

$I_0: S' \rightarrow \cdot S$   
 $S \rightarrow \cdot X S X$   
 $S \rightarrow \cdot c$   
 $X \rightarrow \cdot a$   
 $X \rightarrow \cdot b$

$I_3: S \rightarrow c \cdot$

$I_4: X \rightarrow a \cdot$

$I_5: X \rightarrow b \cdot$

$I_1: S' \rightarrow S \cdot$

$I_6: S \rightarrow X S \cdot X$

$X \rightarrow \cdot a$

$X \rightarrow \cdot b$

$I_2: S \rightarrow X \cdot S X$   
 $S \rightarrow \cdot X S X$   
 $S \rightarrow \cdot c$   
 $X \rightarrow \cdot a$   
 $X \rightarrow \cdot b$

$I_7: S \rightarrow X S X \cdot$

## Problem 6 [12 pts total]

Consider the following Cool-like classes (function bodies ellided):

```
class A {
  a : Int;

  g() : Int { . . . }
  h() : Int { . . . }
}

class B inherits A {
  b: Int;

  j() : Int { . . . }
}

class D inherits A {
  c: Int;

  g() : Int { . . . }
  k() : Int { . . . }
}
```

In your answers to the following questions, assume that A is a base class (not inherited from Object):

**Part 1.** Show the layout of the dispatch tables for classes A, B, and D. You may denote a pointer to the function code for function  $f$  defined in class  $C$  as  $Cf$ .

**Part 2.** Show the layout of objects of classes A, B, and D. You may denote storage for an attribute with its attribute name.

## Problem 7 [6 pts total]

Complete the following type checking rule for the cool assignment expression.

$$O(\text{Id}) = T_0$$

$$\frac{}{O, M \vdash \text{Id} \leftarrow e_1 :}$$

## Problem 8 [5 pts]

Given the following three-address code show the basic blocks and draw the flow graph. You don't need to rewrite every statement; use statement numbers to identify statements in a block.

```
(1)  b := c
(2)  i := 0
(3)  t := 4 * i
(4)  u := a[t]
(5)  v := u * u
(6)  w := 4 * i
(7)  a[w] := v
(8)  if v > d goto 13
(9)  i := i + 1
(10) if I < b goto 3
(11) x := c + d
(12) e := x
(13) y := c + b
(14) f := y
```

## Problem 9 [10 pts]

Consider the following basic block:

```
t1 = b + c
t2 = 7 + d
t3 = t1 * t2
t4 = b + c
t5 = 7 + d
t6 = t4 * t5
t7 = t3 + t6
d = t7
```

**Part A. [5 pts.]** Show the DAG representation of this basic block.

**Part B. [5 pts.]** Generate three-address code from the DAG you made in part A using the heuristic ordering discussed in lecture and the book. Assume that at the end of the basic block  $t_1$ - $t_7$  are dead, and  $b$ - $d$  are live.

Identify (A and B) and circle your final answers.

## Problem 10 [17 pts]

Consider the following new expression to be added to the Cool syntax:

```
switch expr0 {  
  case value1 : expr1;  
  case value2 : expr2;  
  ...  
  case valuen : exprn;  
  default : exprn+1  
}
```

This is a switch expression, which works basically like the switch statement in C/C++, but with the following differences. There must be at least one `case` part in a `switch` statement. Only *one* of the cases will be executed (doesn't fall through to other ones), the `default` is not optional, and only one `value` may be listed for each expression. `expr0` is type `Integer`, and the `valuei` are integer constants. The resulting value of the whole switch expression is the value of `expr0`.

**Part A [5 pts.]** Write a grammar for the `switch` expression. You will start with “E →”.

## Problem 10 (cont)

**Part B [12 pts.]** Write a code generation function for `switch`. Assume the following about the runtime environment and the form of your answer:

- There is a frame pointer `$fp`, stack pointer `$sp`, and accumulator `$a0`
- Use 32-bit integers (not integer objects). Assume the rest of the compiler can handle that (i.e., for doing subexpressions).
- Temporaries are pushed on the stack, not stored in the frame.
- The net effect of evaluating any `E` leaves the result of `E` in `$a0`, and leaves `$sp` and `$fp` unchanged, and may overwrite any other registers.
- You may use MIPS assembly syntax if you know it. Otherwise, you may use the following slightly higher-level pseudo-mips, which has simpler syntax, and allows you avoid the step of loading your operands from memory into a register before using them in an arithmetic operation.

```
x := y           For loading, storing, and moving between registers
x := y op z      Where op is one of +, -, *, /, %, "and", "or"
x := 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: O($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)
in addition ``y'' and ``z'' may also be
    immediate value (i.e., constant)
```

- \*\*\* This can be "pseudo" code-generation code where you can refer to parts of the construct as they are shown as an "argument" to the `cgen` function. However, show actual code generated in calls to `cout <<`.
- \*\*\* You may use a function `newLabel()` to get a unique label (assume it returns a `char *`).
- \*\*\* You must use a loop in terms of `i` for generating the code for the different cases. Pseudo-code such as

```
for (i = 1; i <= n; i++) {
    do stuff with valuei and expi
}
```

\*\*\* = different from sample exam.

## Problem 10 (cont)

```
cgen(switch expr0 {  
    case value1 : expr1;  
    . . .  
    case valuen : exprn;  
    default : exprn+1 }) =
```