CSE 582 Autumn 2002 Exam Sample Solution

Question 1. (10 points) Regular expressions.
Describe the set of strings that are generated by the each of the following regular expressions.

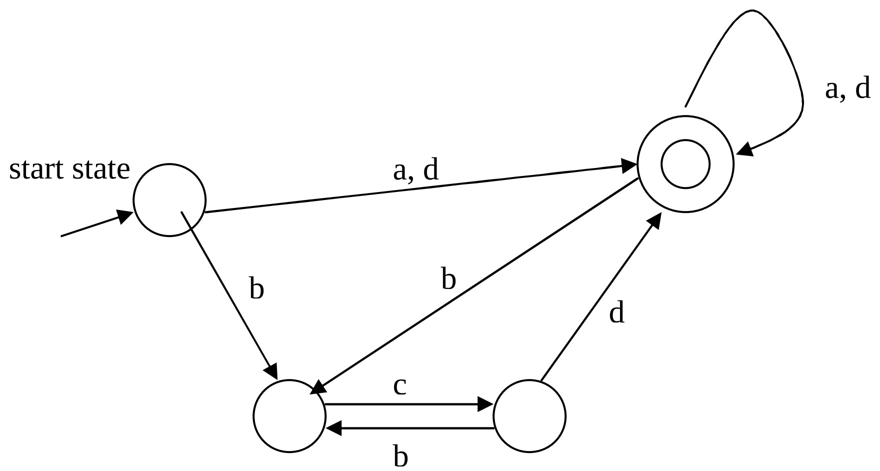
a) (a | (bc)* d)+

*One or more of the string a or the string d preceded by 0 or more pairs of the string bc. (Note that * has highest precedence, concatenation is next, and | has lowest precedence. If your answer was wrong because of mistake in precedence levels, you received most of the credit.)*

b) ((0 | 1)* (2 | 3)+) | 0011

Either the string 0011, or 0 a string consisting of 0 or more 1's and 0's, followed by a string consisting of 1 or more 2's and 3's.

Question 2. (10 points) Draw a deterministic finite automata (DFA) that recognizes strings generated by (a | (bc)* d)+. You do not need to use the formal algorithms for converting regular expressions to a NFA then a DFA (although those algorithms might provide some insight into what's needed). Just draw a suitable diagram.



CSE 582 Autumn 2002 Exam Sample Solution

Question 3. (10 points) A new programming language is being designed that includes decimal numbers with no exponent. A decimal number has a period to separate the integer and fractional parts of the number and **must** have at least one digit both before and after the decimal point. Furthermore, a decimal number **must not** have any superfluous leading or trailing 0's. Some examples of *legal* decimal numbers are 3.14, 0.01, 17.0, 0.0, 123.00321. Some examples of *illegal* decimal numbers are 00.0, .5, 17, 17., 003.01, 0.0012300 (this last example is illegal because of the trailing 0's; the 0 before the decimal point is required).

Write a regular expression (or collection of regular expressions) to generate legal decimal numbers.

(0 | [1-9][0-9])* `.` *(0 | [0-9]*[1-9])*

Question 4. (10 points) Suppose we have the following fragment of a Java program (including some language operators that are not included in JFlat)

```
while (xyzzy >= thing) // repeat until thing is big enough
    { x += y;
     thing++; }
```

List in order the tokens that would be produced by a scanner when reading this input. (Use any reasonable set of token names; just be sure they are clear and descriptive.)

WHILE LPAREN ID(xyzzy) GEQ ID(thing) RPAREN LBRACE ID(x) PLUSASSIGN ID(y) SCOLON ID(thing) PLUSPLUS SCOLON RBRACE

Notes: += and ++ are single tokens (principle of longest match)

Identifier tokens must carry some additional information to specify the particular identifier.

Comments, return, and white space are not tokens – the scanner should filter these out.

CSE 582 Autumn 2002 Exam Sample Solution

Question 5. (12 points) The standard programming language grammar for statements is ambiguous because of the dangling else problem.

stmt ::= `if ( `*expr*` ) `*stmt* | `if ( `*expr*` ) `*stmt* `else` *stmt* | `while ( `*expr*` ) `*stmt* | `S`

Give an unambiguous grammar that defines all of the above statements and correctly handles the dangling else problem in the grammar. (Hint: introduce multiple productions for statements and divide the productions into two categories: those that might end in an `if` statement with no else clause – a "short if" – and those that definitely do not.)

The key idea here is to be sure that the statement following the condition and preceding the else in a full if statement cannot itself be an if statement with no else part.

stmt	::= *stmtNoTrailingSubstmt* | *ifStmt* | *ifElseStmt* | *whileStmt*
stmtNoTrailingSubstmt	::= S
stmtNoShortIf	::= *whileStmtNoShortIf* | *ifElseStmtNoShortIf* | *stmtNoTrailingSubstmt*
ifStmt	::= if (*expr*) *stmt*
ifElseStmt	::= if (*expr*) *stmtNoShortIf* else *stmt*
ifElseStmtNoShortIf	::= if (expr) *stmtNoShortIf* else *stmtNoShortIf*
whileStmt	::= while (*expr*) *stmt*
whileStmtNoShortIf	::= while (*expr*) *stmtNoShortIf*

Notes: Factoring the grammar to change

stmt ::= if (*expr*) *stmt* | if (*expr*) *stmt* else *stmt*

into

stmt ::= if (*expr*) *stmt ifTail*
ifTail ::= else *stmt* | ε

might be useful for preparing the grammar for a parser generator, but it doesn't solve the dangling else problem. It is still possible to generate two distinct derivations (parse trees) for the statement if (*expr*) if (*expr*) *stmt* else *stmt.*

The more subtle problem that many people missed is that the production for a while loop needs to be handled so it cannot generate a short if before the else part of an if statement. There is a dangling else problem if that isn't fixed: if (*expr*) while (*expr*) if (*expr*) *stmt* else *stmt.*

CSE 582 Autumn 2002 Exam Sample Solution

Question 6. (20 points) Grammars and LR parsing.

Consider the following grammar

> *s* ::= *expr* $
> *expr* ::= a
> | a *subs*
> *subs* ::= [*expr*]
> | [*expr*] *subs*

(In the first production, $ represents the end of file)

a) (3 points) What are the terminals and non-terminals of this grammar?

Terminals:

a [ ] $ (full credit if you didn't include $)

Non-terminals:

s expr subs

b) (3 points) Describe in English the set of strings generated by this grammar

Expressions consisting of identifier a followed by 0 or more bracketed expressions, i.e., scalar and array elements in C/C++/Java.

c) There is no part c.

(continued next page)

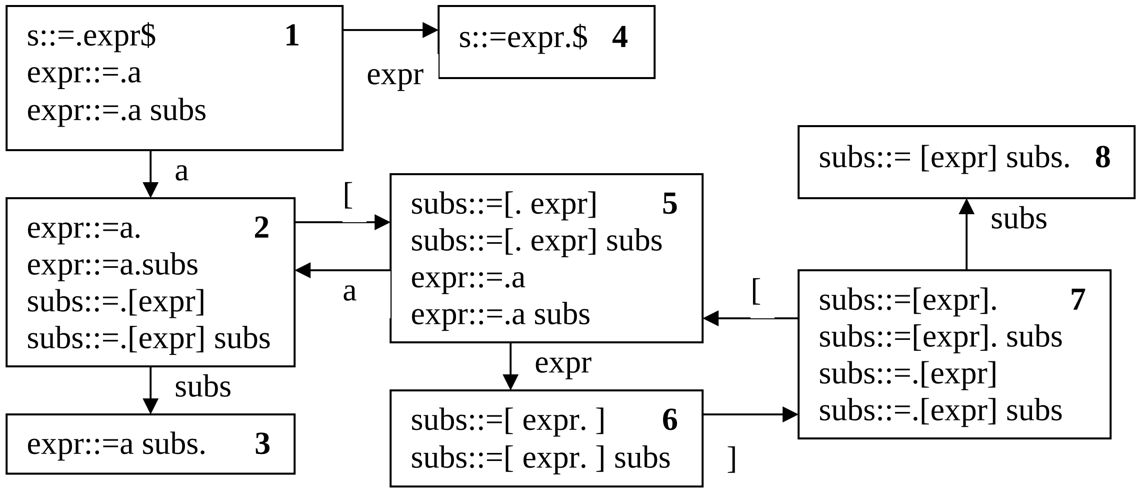
CSE 582 Autumn 2002 Exam Sample Solution

Question 6. (Cont) LR parsing.

(Grammar repeated for reference)

s ::= *expr* $
expr ::= a | a *subs*
subs ::= [*expr*] | [*expr*] *subs*

d) (12 points) Construct (draw) the LR(0) state machine for this grammar. You do **not** need to write out the parser GOTO and ACTION tables, or compute FIRST and FOLLOW sets. Just draw the state machine. Be sure to show the sets of items in each state.



State diagram added for information – not required in your solution.

state	a	[	]	$	s	expr	subs
1	s2					g4	
2	r2,	r2,s5	r2	r2			g3
3	r3	r3	r3	r3			
4				acc			
5	s2					g6	
6			s7				
7	r4	r4, s5	r4	r4			g8
8	r5	r5	r5	r5			

e) (2 points) Is this grammar LR(0)? Why or why not?

No. There are shift/reduce conflicts in states 2 and 7 on input [. (This grammar is SLR(1), however.)

CSE 582 Autumn 2002 Exam Sample Solution

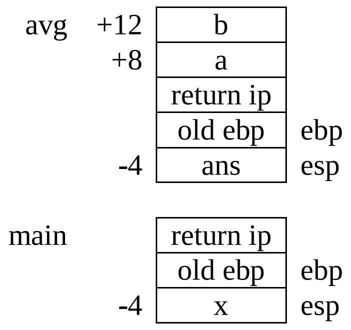
Question 7. (18 points) x86 hacking.

Consider the following C main program and function definition:

```
int avg(int a, int b) {
   int ans;
   if (a == b) {
      return a/2;
   } else {
      ans = (a + b) / 2;
      return ans;
   }
}

void main() {
   int x;
   x = avg(17,42);
}
```

(a) (5 points) Draw a picture showing the layouts of the stack frames for methods `avg` and `main`. This should show the stack layout immediately after the prologue code of each function is executed, just before execution of the first statement of the function body (i.e., after the stack frame has been allocated). Be sure to show where registers `ebp` and `esp` point, and the location and offsets from `ebp` of each parameter and local variable.



Question 7. (cont)

(b) (13 points) Translate both functions `avg` and `main` into x86 assembly language. You do not need to slavishly imitate the code shapes described in class – straightforward x86 code is fine as long as it is correct, uses the registers properly, and obeys the x86 C language conventions for stack frame layout, function calls, and so on. It will help us grade your answer if you include the source code as comments near the corresponding x86 instructions

Your answer had to preserve the semantics of the original program to receive full credit. No points were deducted in avg if the right result was returned without storing a value in ans, however, a store to x should have been included in main.

One subtle point: several people tried to optimize the division operation by replacing cdq/idiv with a shr (shift right) instruction. This works fine for positive operands, but it has different rounding behavior for negative numbers. Shifting truncates towards -∞, while idiv truncates towards 0. That makes a difference for negative odd numbers.

```
avg:  push  ebp          ; function prologue
      mov   ebp,esp
      sub   esp,4
      mov   eax,[ebp+8]  ; if (a==b)
      cmp   eax,[ebp+12]
      jne   L1           ; jump false
      mov   eax,[ebp+8]  ; move a to eax
      cdq                ; divide
      idiv  2            ; result in eax
      jmp   L2           ; jump to common code for return
L1:   mov   eax,[ebp+8]  ; else ans = (a+b)/2
      add   eax,[ebp+12]
      cdq
      idiv  2
L2:   mov   esp,ebp      ; common code for exit
      pop   ebp
      ret

main: push  ebp          ; prologue for main
      mov   ebp,esp
      sub   esp,4
      push  42           ; push args
      push  17
      call  avg          ; call function
      add   esp,8        ; pop args
      mov   [ebp-4],eax  ; store x
      mov   esp,ebp      ; return
      pop   ebp
      ret
```

CSE 582 Autumn 2002 Exam Sample Solution

Question 8. (10 points) Code Shape

Several languages in the ALGOL family had a loop statement designed strictly for counting loops. The syntax for a specific example is:

`for` *var* `:=` *expr1* `to` *expr2* `by` *expr3* `do` *statement*

The semantics of the `for` statement are to repeatedly execute the *statement* that is the loop body with *var* taking on values starting at *expr1*, continuing while the value in *var* is less than or equal to *expr2*, and incrementing *var* by *expr3* after each execution of the loop body. All three expressions (*expr1*, *expr2*, and *expr3*) are evaluated **only once before** the first iteration of the loop. If *expr1* is initially greater than *expr2* then the loop body is not executed. The final `by` *expr3* clause is optional; if it is omitted, `by 1` is assumed instead.

Describe the code shape that could be used to implement this statement on the x86 processor, in the same style that we've used to show code shapes for other programming language constructs. Include the instructions and labels needed to implement the loop and be sure it is clear where the compiled code for the various expressions and the loop body would appear. If you introduce any temporary variables, be sure it is clear where they are stored (where in registers and/or memory).

Two temporaries are needed to store the values of expr2 and expr3 during execution of the loop. In this example, they are pushed on the stack This solution shows a straightforward code sequence with the loop test at the top. That would usually be optimized to put the test at the bottom of the loop There are also optimizations that could be done to avoid loading the control variable immediately after storing it, etc. These are omitted for clarity.

Note: several answers assumed that particular registers like eax and ecx could be used as temporaries throughout the loop to hold expr2 or expr3. Given the code shapes we've discussed, that isn't a reasonable assumption, because code generated for a statement or expression cannot be assumed to leave these registers alone.

```
    < code for expr1 >
    mov   [ebp+offset_var],eax      ; store in control variable
    < code for expr2 >
    push  eax                       ; push expr2 on stack
    < code for expr3 >
    push  eax                       ; push expr3 on stack (esp+0); expr2 now at esp+4
test:                               ; top of the loop – test condition here
    mov   eax,[ebp+offset_var]      ; load control variable
    cmp   eax,[esp+4]               ; compare to expr2
    jnle  done                      ; done if greater
    < code for statement >
    mov   eax,[ebp+offset_var]      ; load control variable
    add   eax,[esp]                 ; add expr3
    mov   [ebp+offset_var],eax      ; store control variable
    jmp   test                      ; iterate
done:
    add   esp,8                     ; pop temporaries from stack
```