UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 164 **P.N. Hilfinger**
Spring 2005

CS 164: Midterm

Name:_____ Login: _____.

You have two hours to complete this test. Please put your login on each sheet, as indicated, in case pages get separated. Answer all questions in the spaces provided on the exam paper. Show all work (but be sure to indicate your answers clearly.) The exam is worth a total of 35+ points (out of the total 200), distributed as indicated on the individual questions.

You may use any notes, books, or computers you please – anything inanimate. We suggest that you read all questions before trying to answer any of them and work first on those about which you feel most confident.

You should have 5 problems on 11 pages.

1. _____/8
2. _____/10
3. _____/
4. _____/9
5. _____/8

TOT _____/35

1. [8 points] Suppose that string literals consist of one or more “stringlets” separated from each other by whitespace (one or more characters, each one of which is either a blank or a newline). Each stringlet consists of any sequence of characters (including newlines) between quotes (“), but with any occurrence of a quotation mark inside the string doubled. For example, “” and “””” are valid stringlets – the first denoting the empty string and the second denoting one quotation mark. Thus the following are valid string literals:

1. “Hello, “ “world.”
2. “I said, “”Hi””””
3. “These are
   two lines”
4. “These are
   “
   “two” “ “ “lines”

a. Give a regular expression for these literals, using Lex notation (including definitions, if you want).

b. Produce the simplest DFA (yes, it must be deterministic) you can for these literals.

2. [10 points] In the following, do not worry about syntax trees or semantics actions; just consider the language being recognized. Symbols in single quotes or in all-upper-case are terminal symbols. It should *not* be necessary to build LALR(1) machines to answer any of these questions!

a. I happened to have an LL(1) parser generator and put the following grammar through it (**Prog** is the start symbol):

Prog -> Stmts '-|'
Stmts -> ε | Stmt Stmts
Stmt -> DO Stmts UNTIL Expr ';'
Stmt -> UNTIL Expr DO Stmts END ';'
Stmt -> ID '=' Expr
Expr -> Term Expr2
Expr2 -> ε | '+' Expr
Term -> ID Arg
Term -> '(' Expr ')'
Arg -> ε | '(' Expr ')'

The parser generator told me the grammar is not LL(1). Why not? In your answer include an input that illustrates the problem.

b. Consider the following grammar, inspired by a widely used programming language. Symbols that don't ever appear on the left of '->' are terminals.

Stmt -> Dcl | Exp
Dcl -> ID Dtor '=' Exp
Dtor -> SDtor | '*' Dtor
SDtor -> ID | SDtor '(' ')' | '(' Dtor ')'
Exp -> ID | ID '(' Exp ')' | '(' Exp ')'

Assuming we want a LALR(1) parser, why do we get a reduce/reduce conflict on the ')' token? In particular, is it because the grammar is ambiguous? Give an example of an input that runs into the error (that is, that puts the parser in a state with a conflict).

c. Consider the following Java (not C) program fragment.

```
class C {
    int g (int y, int z) {
        If (y > 0) {
            int r = y - z;
            return f (r – 1);
        } else {
            int q = z * y;
            return x + q;  //<<<
        }
    }

    int x = 2;

    int f (int x) {
        return g ( x+1,2);
    }

    String h (sting x) {
        return x + g (Integer.parseInst (x), 1);
    }
}
```

what would the compiler's symbol table look like while it is processing the body of **g** at the <<< comment?

d. For the same program as in part (c), would the behavior of the class change at all if Java were dynamically scoped? If so, then give an example (i.e. of a method call on an instance of C) that would exhibit the difference. Otherwise explain why there is no difference.

e. For the same program as in part (c), and again assuming normal Java scoping rules, would the behavior change if Java were dynamically typed? Why?

3. [1 point] Where, according to the poet, did one find “a miracle of rare device”?

4. [9 points] Consider the following grammar:

start -> prog -| {_____}
prog -> ε {_____}
prog -> prog stmt {_____}
stmt -> assign ‘;’ {_____}
stmt -> FOR NUM DO prog finally OD ‘;’ {_____.
_____}
finally -> FINALLY prog {_____}
finally -> ε {_____}
assign -> ID ‘=’ expr {_____}
expr -> ID {_____}
expr -> NUM {_____}
expr -> assign {_____}

I am interested in finding the total number of assignments made to the variable ‘x’ in a given program. A **for** loop executes its statements as many times as its counter instructs. The **finally** clause executes iff the counter is greater than zero. For example the answer for each of the following three programs is 12:

```
a = x = b;
for 10 do
        a = x; x = c; g = o = o = d;
od;
x = 5;
```

_____.

```
for 3 do
        for 3 do
                b = x = 4; l = u = c = k;
        finally
                x =a; c = d; d = x;
        od;
od;
```

```
a = x =b
for 10 do
        a = x; x = c; g = o = o = d;
od;
for 0 do

x = a; a = x;
finally
    x = 3; x = 1; x = 4;
od;
x = 5;
```

Questions begin on next page

a. Using Bison notation, fill in the actions above to compute the right answer (that is so that the semantic value assigned to the start node is the desired number). Pretend that types have been declared for all the symbols: ID has type String and everything else is an integer. If there are nonterminals that don't need values, don't bother to assign them. Your actions must have no side effects (no global variables in particular).

b. Fill in the recursive-descent compiler on the next page so that the input program gets checked for syntactic correctness and the right number gets printed. Be careful: the grammar is not LL(1); you can change it as needed just so long as you end up recognizing the same language and get the numbers right. Assume the following functions are available for your use:

next() returns the syntactic category of the next (as yet unprocessed) symbol of the input as one the values

'=' ID NUM FOR DO OD ';' '-|'

name() returns the name (a String) of the next token whenever next () == ID

value() returns the integer value of the next token whenever next() == NUM

scan(T) checks that next () is T and reports and error if not. It then advances to the next token.

ERROR() reports an error.

Write your program on the next page without using global variables. All assignments should be local variables only. Do not introduce any new types.

Here is the parser skeleton. Remember no global variables; assign only to local variables; do not introduce new types. If needed, however you can introduce more functions (do mention somewhere what they parse).

```
void start() {
        printf ( “Number of assignments to x in program = %d\n”, prog());

}

int prog(){









}


int stmt () {









}


int _finally () {









}
```

Continued on the next page

Continued from the previous page

```
int assign() {




}


int expr () {




}
```

6. [8 points] A certain language has the following terminal symbols:

   @ # i -|

(the -| symbol is an end-of-file). A shift-reduce parser for the language processes the string

i # i @ i -|

And performs the following actions

1. Shift #
2. Reduce 1 symbol (from the stack), producing (the non terminal symbol) P
3. Shift i
4. Reduce 2 symbols, producing E
5. Shift #
6. Reduce 0 symbols producing P
7. Shift i
8. Reduce 2 symbols producing E
9. Shift @ Then shift i
10. Reduce 3 symbols producing E
11. Reduce 3 symbols producing E
12. Shift -| accepting the string

(When we say “reduce n symbols producing X” above, we main “remove n symbols (terminals or non-terminals) from the stack, and then push the non-terminal X”). Now for the questions:

a. What is the parse tree for the given string?

b. Fill in the following shift-reduce table with a plausible set of entries *for the given input*. No, we are not interested in the LALR(1) table necessarily, just *any* table that would behave as indicated on the given input (how it would (mis)behave on some other input is completely irrelevant). The starting state is 0; otherwise, you are free to choose your own state numbers. For each rn (reduce) entry that you have, provide the appropriate production on the right. Feel free to leave rows blank if you don't need them. Likewise, you don't need to use all of the rn entries we've given you. **Important:** *there is more than one correct answer to this question!*

Action/Goto						
State	@	#	*i*	-|	*E*	*P*
0.						
1.						
2.						
3.						
4.						
5.						
6.						
7.						
8.						
9.						
10.						
11.						

r1_____.

r2_____.

r3 _____.

r4_____.

r5_____.

r6_____.

r7_____.

r8_____.