UNIVERSITY OF CALIFORNIA

Department of Electrical Engineering

and Computer Science

Computer Science Division

CS164

Paul Hilfinger

Spring 1999

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

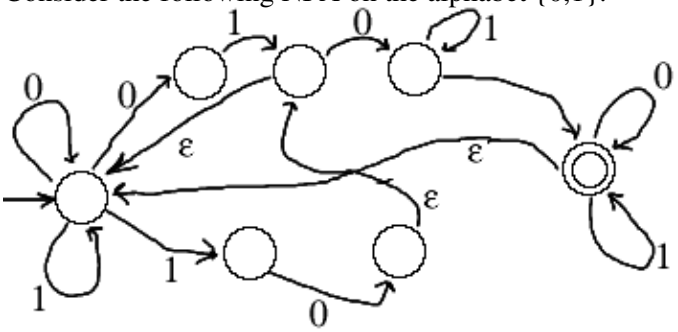
Problem #1 [9 points]

A certain language has the following terminal symbols: @ ># i -|
(the -| symbol is an end-of-file). A shift reduce parser for this language processes the strin
@ i >i # @ i -|
and performs the following actions:
1. Shift @. Then shift i
2. Reduce 0 symbols (from the stack), producing (the non-terminal symbol) N. Then reduce 2 symbols, producing E
3. Shift >. Then shift i. Then shift #.
4. Reduce 1 symbol. producing N. Then reduce 2 symbols. Producing E
5. Shift @. Then shift i
6. Reduce 0 symbols, producing N. Then reduce 2 symbols, producing E. Then reduce 3 symbols producing E. Then reuce 4 symbols, producing E
7. Shift -|. accepting the string.

(When we say "reduce n symbols" above we mean n terminal and nonterminal symbols, not states. For example, on page 11 of Handout #3, there is a parser stack that looks like this:
$_0$i$_2$. The i counts as one symbol. we ignore the 0 and 2 for this problem.

Problem #2 [9 points]

Consider the following NFA on the alphabet {0,1}:



a. Describe as succinctly as possible (in English) the language recognized by this NFA (my one sentence answer to this question, for example, is less than 10 words long). WARNING: Some of the transitions are superfluous.
b. Write the simplest regular expression you can that describes the same language. Use Lex notation. (No we did not cover a general method of doing this in class. Devise a regular expression directly from your description in (a).)
c. Write the simplest BNF grammar you can that describes the same language.

Problem #3 [1 point]

Where do poppies blow between the crosses, row on row?

Problem #4 [9 points]

Consider the following grammar:
start ->prog -|
prog ->ε
prog ->stmt prog
stmt ->assign
stmt ->IF expr THEN prog else FI ';'
else ->ELSE prog
else ->ε
assign ->ID '=' expr ';'
expr ->ID
expr ->expr '+' expr
I am interested in finding the maximum number of '+' operations in any single expression (expr) of a given program. For example the two programs

```
x = a + b;
if e + f + a then
   e = f
fi;
```

and

```
if e + r then
   x = y; q = a + b + c;
else a = b+c;
    c=a+d; d = a+e;
fi;
```

the answer should be 2 for both programs (this assumes that ID denotes 1-character identifiers.) Fill in teh

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, jkust so long as you end up recognizing the same language and get the right numbers. Assume that the following functions are available for your use:

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

+ = ID IF THEN ELSE FI ; -|

The lexer attacheds no semantic information to the tokens.

scan (T)checks that next() is T, and reports an 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 to 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 intropduce new types. This is pseudo-C++, so don't worry about declaring functions before use.

```
void start(){
  printf ("Maximum operators in any expression = %d\n",prog());
}

___ prog (){

}
___ stmt (){

}
___ _else(){

}
___ assign(){

}
___ expr (){

}
```

Problem #5 [8 points]

In the following do not worry about syntax trees or semantic actions; just consider the lanauge 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:
Expr ->Term '+' Factor | Factor
Term ->Prefix Expr
Prefix ->ε| '-'
Factor ->ID | '(' Expr ')'
The parser generator told me the grammar is not LL(1). Why not?

b. In the language Alphard, x.y could be written y(x). Assuming we want a LALR (1) parser, why do we get a reduce/reduce conflict on the ')' token int eh follwoing *unambiguous*grammar? Give an example of an input that runs into the error

Stmt ->Assign | Call
Assign ->Var ':=' Expr
Var ->ID | ID '(' Var.')'
Expr ->ID | Call | '(' Expr ')' | Expr '+' ID
Call ->ID '(' ExprList.')'
ExprList ->Expr | ExprList ',' Expr

c. Consider the following C++ program fragment

```
int x = 2;
int g (int y) {
    if (y>0)
        return g (y-1);
     return x+y;
}
void f (int x) {
      g (x+1);
}
```

Show the environment (as in 61A-style environment diagrams) during execution of the expression f(0) at the point where g iis executing and y is 0. Show the environment you'd have at the same place if C++ used dynamic scope. [Please label which diagram is which clearly].

d. For the same program as part (c), what would the compiler's symbol table look like while it is processing the body of g?