UNIVERSITY OF CALIFORNIA

Department of Electrical Engineering

and Computer Sciences

Computer Science Division

P. N. Hilfinger

CS164
Fall 1997

CS 164: Midterm

Name: _____ Login: _____

Please do not discuss the contents of this test with anyone before the evening of Friday, 10 October.

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 space 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 of 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 8 pages.

1. _____/9

2. _____/9

3. _____/

4. _____/9

5. _____/8

TOT. _____/35

Login:

1. [9 points] A certain language has the following terminal symbols:

@ * # i /

A shift-reduce parser for this language processes the string

i * i # @ i /

and performs the following actions:

1. Shift **i.**

2. Reduce 0 symbols (on the stack), producing (the non-terminal symbol) **N**. Then reduce 2 symbols, producing **T**. Then reduce 1 symbol, producing **F**.

3. Shift *****. Then shift **i**. Then shift **#**.

4. Reduce 1 symbol, producing **N**. Then reduce 2 symbols, producing **T**.

5. Shift **@**. Then shift **i.**

6. Reduce 0 symbols, producing **N**. Then reduce 2 symbols, producing **T**. Then reduce 1 symbol, producing **F**. Then reduce 5 symbols, producing **F**.

7. Shift **/**.

8. Reduce 2 symbols, producing **E** and accepting the input.

Now for the questions:

a. As far as one can tell from this sample, what is the grammar for the language being parsed?

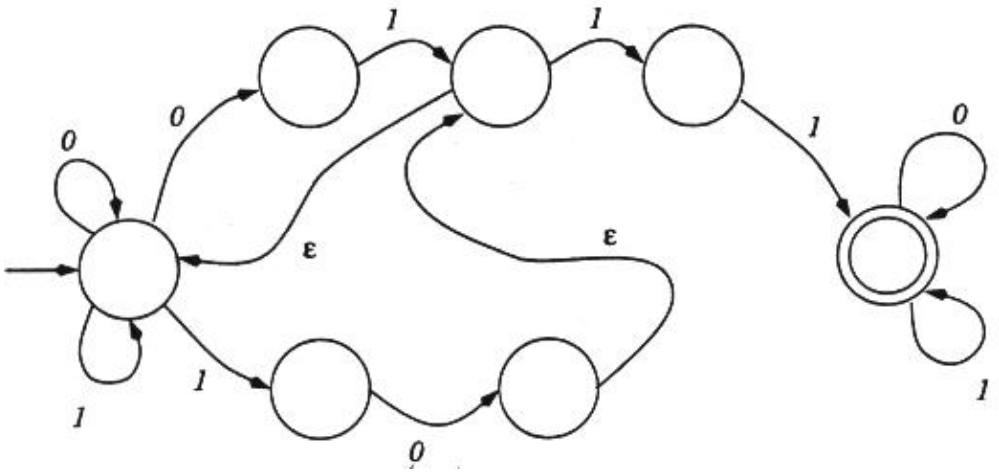
(Continued on next page)

Login:

b. What is the parse tree for this string?

Login:

2. [9 points] Consider the following NFA:



a. Describe as succinctly as possible (in English) the language recognized by this NFA.

b. Write a regular expression that describes the same language. You may use Lex notation.

c. Write a BNF grammar that describes the same language.

Login:

3. [1 point] If q is analytic everywhere within and on a closed contour ∂ in the complex plane and κ is interior to ∂, then what is

$$\frac{1}{2\pi i}\int_{\partial}\frac{q(\gamma)}{\gamma-\kappa}d\gamma?$$

4. [9 points] Consider the following grammar:

```
start → prog ⊣
prog → ϵ  | stmt prog
stmt → assign  |  WHILE expr DO prog OD ';'
assign → ID '=' expr ';'
expr → ID | ID '+' expr
```

I am interested in finding the maximum number of '**+**' operations in any expression (**expr**) of a given program. For example, in

```
x = a + b;
while e + f + a do
   e = f;
od;
```

the answer should be 2 (this assumes that **ID** denotes 1-character identifiers.)

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 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  WHILE  DO  OD  ;  ⊣
```

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.

Login:

Here is the parser skeleton. Remember: no global variables; assign only to local variables; do not introduce new types.

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

Login:

5. [8 points] In the following, do not worry about syntax trees or semantic 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:

```
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 in the following grammar? Give an example of an input that runs into the error. (Warning: is it ambiguous?)

```
Stmt → Assign | Call
Assign → Var ':=' Expr
Var → ID | ID '(' Var ')'
Expr → ID | Call | '(' Expr ')' | Expr '+' ID
Call → ID '(' ExprList ')'
ExprList → Expr | ExprList ',' Expr
```

Login:

c. In the following grammar, there is a shift/reduce conflict on '('. Bison says that the problem is in the following state:

```
Expr  ->  ID .   (rule 10)
Expr  ->  ID . '(' Expr ')'   (rule 11)
Expr  ->  ID . '+' Expr   (rule 12)
Expr  ->  ID . '=' Expr   (rule 13)
```

Identify the bug in the grammar. Give an example of an input that runs into the error (that is, that requires making a shift/reduce decision for a state and symbol that have a conflict). Propose a simple change to the grammar (and the language it describes) to fix the problem (that is, make an informed guess as to what was really intended).

```
Function → TYPE ID '(' Args ')' '{' Stmts '}'
Args → ID | ID ',' Args
Stmts → ϵ | Stmts Stmt
Stmt → IF '(' Expr ')' Stmt ELSE Stmt  |  ExprStmt
ExprStmt → Expr
Expr → '(' Expr ')' | ID | ID '(' Expr ')' | ID '+' Expr | ID '=' Expr
```

d. The following expression grammar makes addition and multiplication left-associative and exponentiation (**) right associative. What causes the shift-reduce conflicts on the symbols +, *, and **? Give an example of an input that runs into one of the errors. (Be careful: TYPE and ID are *distinct* tokens.)

```
Expr → Term | Expr '+' Term
Term → Factor | Term '*' Factor
Factor → Primary | Primary '**' Factor
Primary → ID | '(' Expr ')' | '(' TYPE ')' Expr
```