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

CS 164 **P. N. Hilfinger**
Fall 2011

CS 164: Test #1

Name: _____ Login: _____

You have eighty minutes 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 20+ points (out of the total of 200), distributed as indicated on the individual questions.

You may use any notes or books you please—anything unresponsive. 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 9 pages.

1. _____/5	4. _____/
2. _____/3	6. _____/6
3. _____/6	TOT _____/20

1. [5 points] A certain programming language allows three types of integer literal:

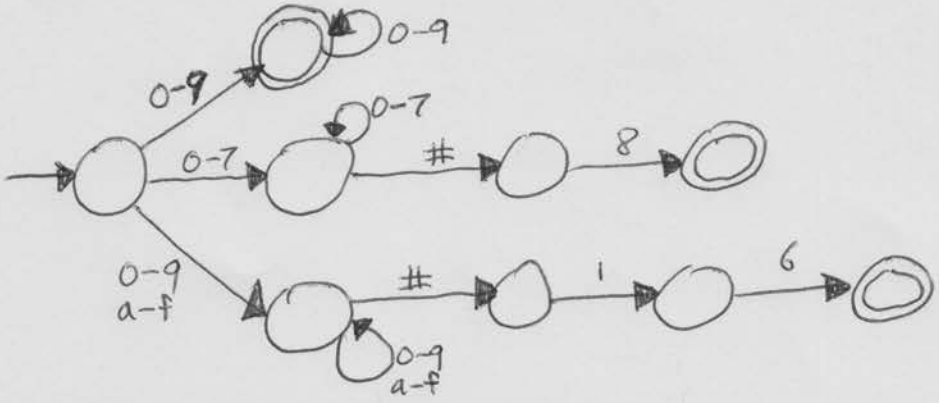
- octal, written with a trailing `#8`, as in `0#8` and `12#8` for 0 and 10;
- decimal, written with no trailing suffix, as in `0` and `185`;
- hexadecimal, written with a trailing `#16`, as in `3f#16` and `e#16` for 63 and 14. The digits representing 10–15 must be lower case (`3F#16` is invalid).

Leading 0s are legal, but do *not* indicate octal in this language.

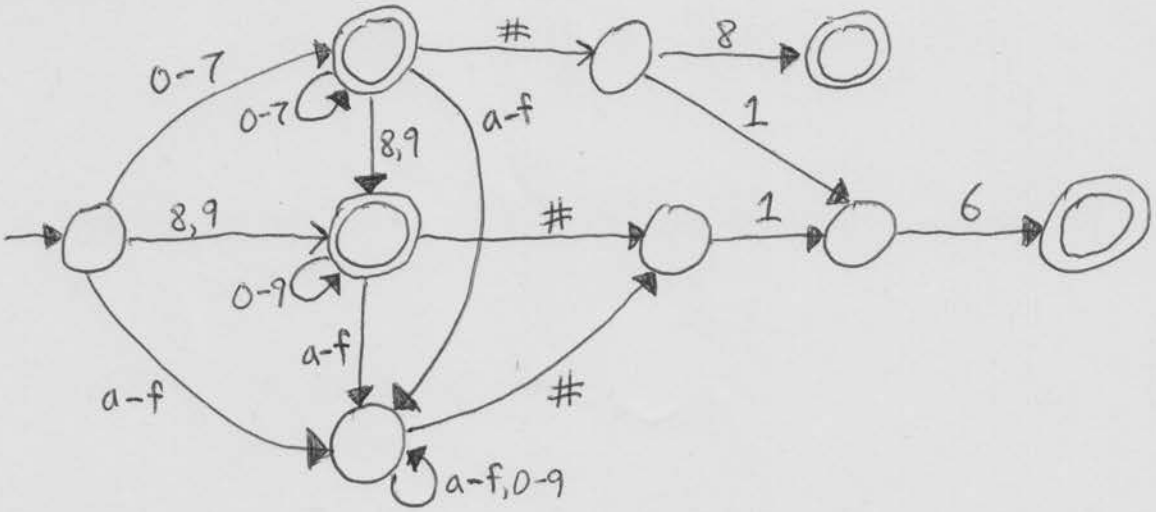
a. Write a regular expression representing these numerals.

$[0-9]+ \mid [0-7]+\#8 \mid [0-9a-f]+\#16$

b. Create the simplest NFA you can that recognizes these numerals. (Any NFA will do; you do not have to use the general construction covered in the notes and lecture).



c. Create a DFA that recognizes these numerals.

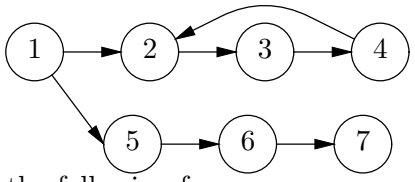


2. [3 points] The table below shows the first column in a chart produced by Earley's algorithm. Fill in the second column (after the input of the terminal symbol denoted 'C'):

C

0	*1*
p: • r ⊢, 0	r: (•, 0
r: • r r, 0	ρ: r • ⊢, 0
r: • C, 0	r: r • r, 0
r: • r '+' r, 0	r: r • '+' r, 0
	r: • r r, 1
	r: • (, 1
	r: • r '+' r, 1

3. [6 points] Consider the problem of describing a directed graph such as:



using a textual language with the following form:

```
vtx 1 -> vtx 2 -> vtx 3 -> vtx 4 -> 2
    & -> vtx 5 -> vtx 6 -> vtx 7
```

The proposed syntax is as follows:

```
%token LAB

%%

prog: graph
    ;

graph: /*empty*/
     | graph vertices
     ;

vertex: "vtx" LAB
       | LAB
       ;

vertices: vertex
        | vertex edges
        ;

edges: edge_list
      | edges "&" edge_list
      ;

edge_list: "->" vertices
          ;

%%
```

The clause `vtx` L introduces a new vertex named L. After this introduction, other occurrences of the same label L denote the same vertex. The '`->`' operator denotes an edge from the vertex on its left to that on its right. The '`&`' operator, as shown in the example, introduces a new chain of edges from a previous vertex in the chain.

Questions begin on the next page.

a. Unfortunately, our parser generator reports conflicts with this grammar. Why?

The grammar is ambiguous.

For example, "1 -> 2 -> 3 & -> 4" has two distinct leftmost derivations. Both begin:

prog ⇒ graph ⇒ graph vertices ⇒ vertices ⇒ vertex edges ⇒ 1 edges

But then we can continue:

1 edges ⇒ 1 edges "&" edge_list OR

1 edges ⇒ 1 edge_list

b. On the next page, give a corrected version of this grammar that resolves these conflicts in such a way that the grammar interprets the example consistently with the diagram. Do not use `%left` or `%right` directives.

c. The *out degree* of a vertex is the number of edges that leave it (going to another vertex). On the next page, supply semantic actions for your grammar from (b) that assign a Python-like map value, M, as the semantic value of the 'prog' non-terminal such that for any vertex label, S, $M[S]$ is the out degree of S. We will not be fussy about notation or type-checking (as long as your solution is clear and makes sense). In particular, assume that semantic values can be anything you want. The semantic value of the LAB terminal symbol will be a string (the label of a vertex). Other semantic values can be lists, maps—anything you find convenient. However, you may *not* use global variables and your actions should have no side-effects (produce new objects as values; don't change old ones). Feel free to invent and use functions on lists and maps just by giving a *clear and concise* description of what they do, without bothering to implement them. You do *not* need to check that all vertex labels used in edges are defined, nor that vertices are defined only once. If there are two edges in the same direction between two nodes, count them as separate edges. Self loops (edges from a vertex to itself) are allowed.

```
prog: graph             {   $$ = $graph                          }
    ;
graph: /*empty*/        {   $$ = {}                              }
     | graph vertices   {   $$ = sum_key_values($graph, $vertices) }
     ;
vertex: "vtx" LAB       {   $$ = $LAB                            }
      | LAB             {   $$ = $LAB                            }
      ;
```

corrected grammar rules go here:

```
vertices: vertex          { $$ = {$vertex : 0} }
        | vertex edges    { (n, out_deg_map) = $edges
                            $$ = sum_key_values( {$vertex_# : n},
                                                 out_deg_map ) }

edges : edge_list         { $$ = (1, $edge_list) }
      | edges "E" edge_list { (n, out_deg_map) = $edges
                              $$ = (n+1,
                                    sum_key_values(out_deg_map,
                                                   $edge_list)) }

edge_list : "->" vertex          { $$ = {$vertex : 1} }
          | "->" vertex edge_list { $$ = sum_key_values({$vertex : 1},
                                                        $edge_list) }
```

4. [1 point]

$$\frac{P(X|Y) \cdot P(Y)}{P(X)} = \underline{\quad P(Y|X) \quad}$$

5. [6 points] On a certain input, a shift-reduce parser performs the following actions (read left to right, top to bottom):

```
s[X]        r3 1->i     s["["]      s[T]
s[A]        r7 0->a     r8 2->a     s["]"]
r5 4->t     s[X]        r3 1->i     r1 0->h
r2 2->h     s["[+"]     s[T]        s["]"]
r6 3->e     r4 3->i     r1 0->h     r2 2->h
r2 2->h     accept
```

Here, '`s[`Y`]`' means "shift the terminal symbol Y," and '`r`K N`->`S' means "reduce by rule #K, popping N symbols and shifting the nonterminal symbol S." The terminal symbols are

```
"["   "]"   "[+"   A   T   X
```

a. Show the input string.

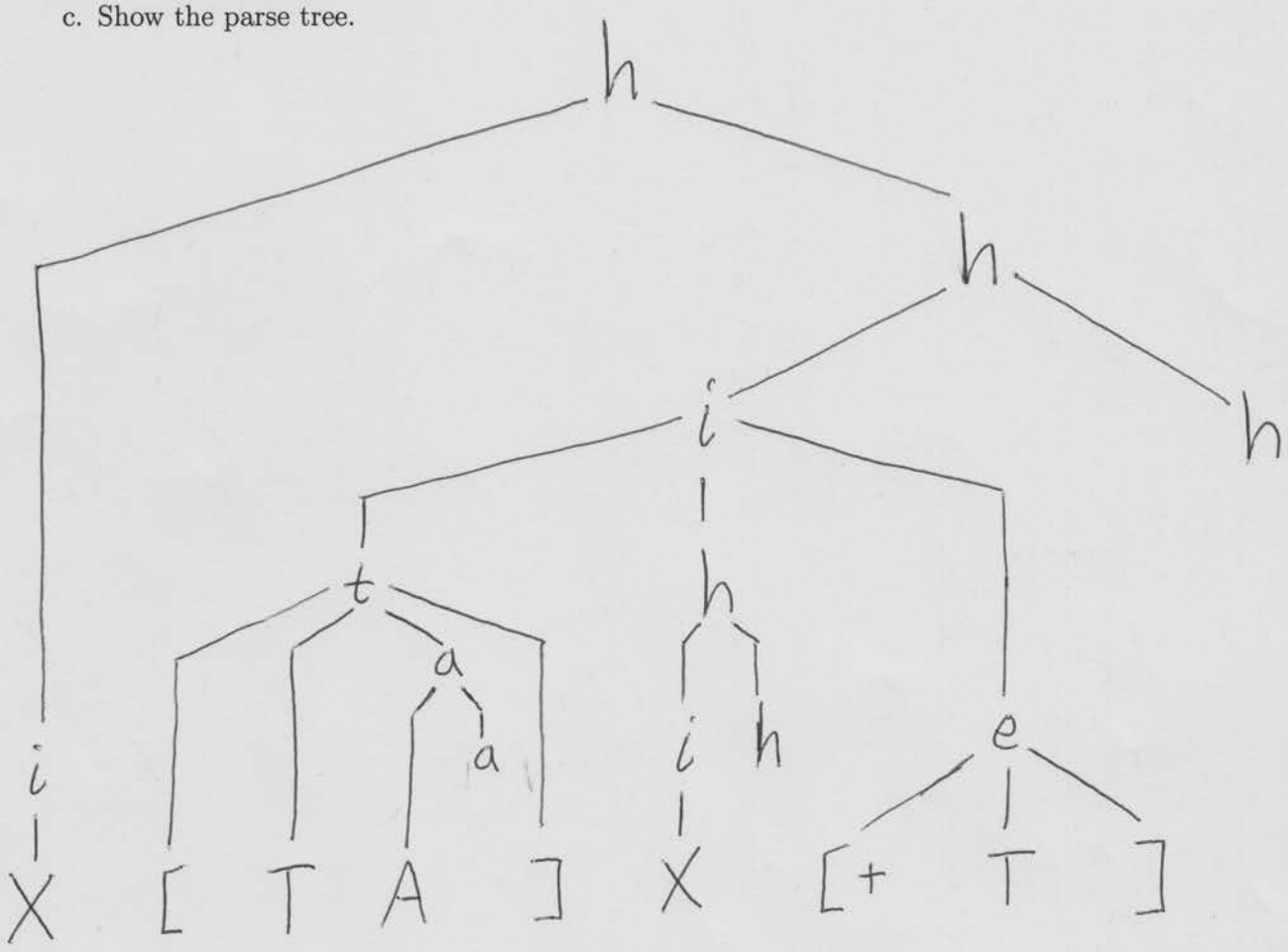
X [T A] X [+ T]

b. Show the first three steps of the rightmost derivation of the input.

h ⇒ i h ⇒ i i h ⇒ i i

Continued on next page.

c. Show the parse tree.



Continued on next page.

d. What are grammar rules #1–#8?

1: $h \to \varepsilon$

2: $h \to i\ h$

3: $i \to X$

4: $i \to t\ h\ e$

5: $t \to [\ T\ a\ ]$

6: $e \to [+\ T\ ]$

7: $a \to \varepsilon$

8: $a \to A\ a$

e. Is this grammar LL(1)? Why or why not? (An informal argument is fine. You need not show the actual calculation of FIRST and FOLLOW, for example. Instead, you can figure them out "by eye" from the grammar and just state what they are, as needed.)

Yes - the grammar is LL(1) because no nonterminal has two productions with conflicting FIRST/FOLLOW sets.

For h:

FIRST $(\varepsilon) = \{\varepsilon\}$

FIRST $(i\ h) = \{X, [\}$

FOLLOW $(h) = \{[+\}$

For i:

FIRST $(X) = \{X\}$

FIRST $(t\ h\ e) = \{[\}$

For a:

FIRST $(\varepsilon) = \{\varepsilon\}$

FIRST $(A\ a) = \{A\}$

FOLLOW $(a) = \{]\}$