CS 164, Spring 2010 **CS 164: Test #1 Solutions** **P. N. Hilfinger**

1. [2 points] Consider the following very simple grammar:

```
p : s ⊣
s : /* Empty */
s : '(' s ')'
```

Suppose that we have a shift-reduce parser for this grammar. Write a regular expression that describes the possible contents of the parsing stack just after a shift or reduce.

Answer:

```
"("+s")"|"("*s?
```

or, if you include the final shifting of '⊣' and reduction to `p`,

```
"("+s")"|"("*s?|s⊣|p
```

2. [1 point] I have a language implementation that involves a FLEX lexer. A new revision of the language standard requires me to double the number of different kinds of token returned by the lexer. What effect will this have on the lexer's speed and why? (Assume that both new and old lexer action rules (the things in curly braces) all take about the same time to execute.)

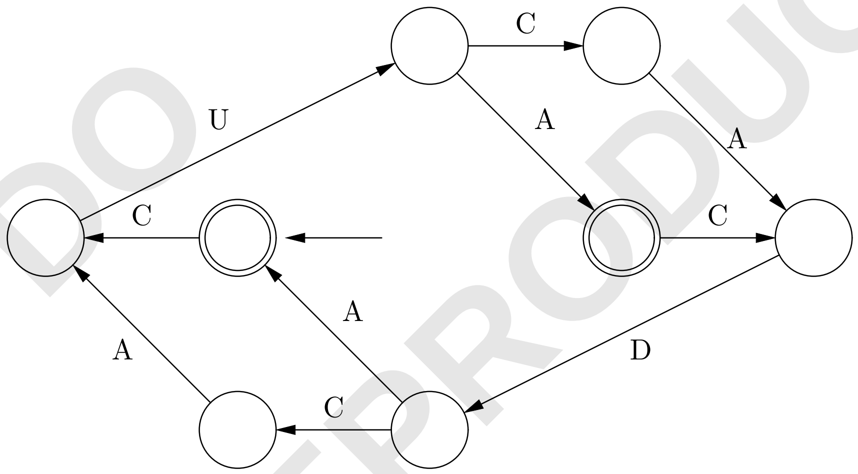
Answer: Very little if any difference. These lexers are table-driven, doing one table lookup for each character. Unless the average length of a lexeme increases, the time spent will therefore tend not to change.

3. [1 point] From a given grammar, I am able to produce two different derivations of a certain string, S. Under what circumstances would this indicate that the grammar is ambiguous?

Answer: If both were the same kind of derivation—both leftmost or both rightmost, for example—the grammar would have to be ambiguous. In general, the derivations would have to expand the same instance of a particular nonterminal in two different ways.

4. [3 points] An elevator travels back and forth between two floors. We can describe its operation in the abstract as a sequence of events. It receives a command (C) to move up or down to the other floor, it starts moving up (U) or down (D), and it arrives at the other floor (A), after which the cycle can repeat. After the elevator starts moving and before it arrives, it can receive a command (C) to return to the floor it just came from, in which case it will start back in the opposite direction after it arrives (A) at the floor it's going to. So in the abstract, we can describe a day on the elevator with a sequence of letters from the alphabet $\{A, C, D, U\}$. Design a DFA that recognizes the legal sequences of events, assuming that the elevator starts on the lower floor (motionless), and ends on either floor with no commands pending. Ignore the possibility of redundant commands (receiving two 'C's that command the same thing before finishing the first). For example, "CUCADACUA" is valid, but "UDA," "CUUA," "CUCA," and "CCUA" are not.

Answer:



5. [1 point] What is the name of the dealer you'd go to for

> ... answers oracular,
> Bogies spectacular,
> Tetrapods tragical,
> Mirrors so magical,
> Facts astronomical,
> Solemn or comical,...?

Answer: John Wellington Wells (Gilbert & Sullivan, *The Sorcerer*).

6. [5 points] The following grammar describes the structure of a tree-like specification language:

```
spec  : block ⊣

block : TAG ID attrs kids

attrs : /* EMPTY */
      | attr attrs

kids  : /* EMPTY */
      | BEGIN blocklist

blocklist :
        END
      | block blocklist

attr  : ID '=' NUM
```

Upper-case names and quoted symbols denote terminal symbols (returned by the lexer). Assume the following are already implemented:

```
# The syntactic category of the next token (ID, BEGIN, END, '=', etc.).
def peek(): ...

# If token is not None, check that peek() == token. Then discard the
# current value of peek() and replace it with the next token in the input.
# Indicate an error if the check fails.
def scan(token = None): ...

# Report error
def ERROR(): ...
```

On the next page, complete a recursive-descent parser for this language. Do recognition only; don't worry about semantic values.

```
def spec():

    block(); scan('⊣')

# Complete the implementation here.
def block():
    scan(TAG); scan(ID); attrs(); kids()

def attrs():
    if peek() == ID:
        attr(); attrs()
# The following two lines are optional---the error could be caught later.
    elif peek() not in [ BEGIN, TAG, END, '⊣' ]:
        ERROR()

def kids():
    if peek() == BEGIN:
        scan(BEGIN); blocklist()
# The following two lines are optional---the error could be caught later.
    elif peek() not in [ TAG, END, '⊣' ]:
        ERROR()

def blocklist():
    if peek() == END:
        scan(END)
    else:
        block(); blocklist()
# or replace the last two lines with
#   elif peek() == TAG:
#       block(); blocklist()
#   else:
#       ERROR

def attr():
    scan(ID); scan('='); scan(NUM)
```

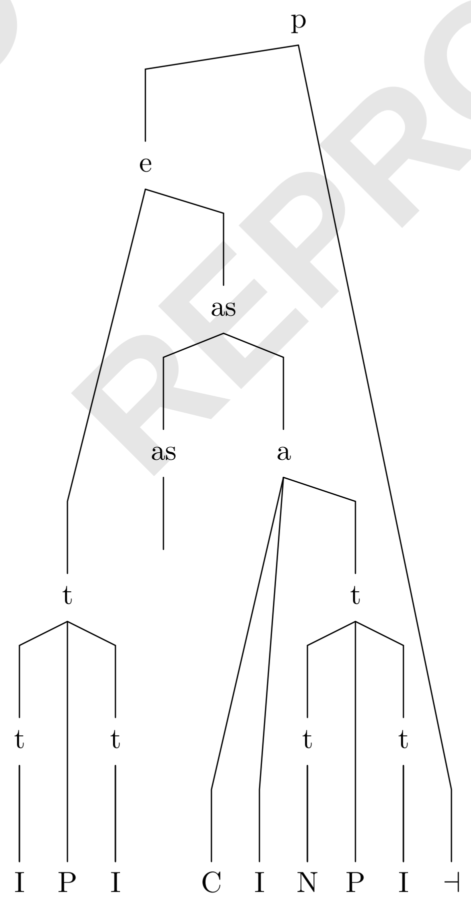
7. [4 points] Consider the following reverse derivation:

```
I P I C I N P I ⊣  ⇐
t P I C I N P I ⊣  ⇐
t P t C I N P I ⊣  ⇐
t C I N P I ⊣  ⇐
t as C I N P I ⊣  ⇐
t as C I t P I ⊣  ⇐
t as C I t P t ⊣  ⇐
t as C I t ⊣  ⇐
t as a ⊣  ⇐
t as ⊣  ⇐
e ⊣  ⇐
p
```

Upper-case letters (and '⊣') are terminal symbols; lower-case letters are nonterminals.

a. Show the parse tree corresponding to this derivation.

Answer:



Problem continues on next page

b. Show as much as you can of the grammar corresponding to this derivation.

Answer:

```
p  : e ⊣
e  : t as
t  : I
t  : N
t  : t P t
as : ϵ
as : as a
a  : C I t
```

c. Is the grammar ambiguous? Why or why not?

Answer: Yes. The production "`t :  t P t`" for example makes parsing "`I P I P I`" ambiguous.

d. Starting with `p`, show the first five derivation steps (the first five '⇒' steps) in the leftmost derivation that corresponds to the same parse tree.

Answer:

`p` ⇒ `e ⊣` ⇒ `t as ⊣` ⇒ `t P t as ⊣` ⇒ `I P t as ⊣` ⇒ ⋯.

8. [4 points] A very primitive text formatter for mathematics allows one to write things like "`x^2+y_{i+1}^{3}-c_{q_{j}}`" and get "$x^2 + y^3_{i+1} - c_{q_j}$". The grammar below gives the syntax for the input language:

```
/* Semantic values are tuples (<BOX>, <WIDTH OF BOX>) */

eqn : text                  { output($1[0]); }


text : /*EMPTY*/            { $$ = (box(), 0); }


     | text item            { $$ = (add ($1[0], $2[0], $1[1], 0), $1[1]+$2[1]); }


item : GLYPH sub0 sup0      { t = add ($1, $2[0], 1, -1)
                              t = add (t, $3[0], 1, 1)
                              $$ = (t, 1 + max($2[1], $3[1])) }

sub0 : /*EMPTY*/            { $$ = (box(), 0) }


     | '_' '{' text '}'     { $$ = $3 }


sup0 : /*EMPTY*/            { $$ = (box(), 0) }


     | '^' '{' text '}'     { $$ = $3 }
```

The terminal symbols are `GLYPH`, '`{`', '`}`', '`_`', and '`^`'. The object that you output is a *box,* a container of printable text, which is defined for you in the form of two functions:

box() returns an empty box.

add(B_1, B_2, x, y), where B_1 and B_2 are boxes and x and y are numbers, returns a new box containing B_1's text, with B_2's text positioned so that its lower-left corner is at coordinates (x, y) relative to B_1's lower-left corner. Thus if B_1 contains the text "`hello␣`" and B_2 contains the text "`world`", then `add(`$B_1, B_2, 6, 1$`)` would output as $\texttt{hello␣}^{\texttt{world}}$.

For each GLYPH The lexical analyzer returns a box containing text of width 1. Superscripts and subscripts are one unit above and below the text they are attached to, respectively. We will not bother with decreasing the font size of subscripts and superscripts for this problem.

Fill in the blanks above to achieve the desired effect. Your semantic actions must not have side effects, except to assign to `$$`. You may, however, use any Python values and data structures you'd like for semantic values (numbers, boxes, dictionaries, lists, etc.).