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

CS 164 **P. N. Hilfinger**
Fall 2006

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 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 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 11 pages.

1.	________/8
2.	________/10
3.	________/
4.	________/9
5.	________/8
TOT	________/35

1. [8 points] In a certain language, string literals are surrounded by (double) quotation marks (`"..."`) and may contain any number (zero or more) of blanks and any of the "graphic" (visible, printing) characters other than quotation marks. In addition, they may contain hexadecimal escape sequences of the form

`["` *hh*`"]`

where *hh* is a pair of hexadecimal digits (0–9, a–f, A–F). Square brackets, by the way, are also valid graphic characters. Thus, the following are valid string literals:

- `""` (the empty string).
- `"He said, ["22"]Hi["22"]!"` (the string "`He said, "Hi"!`", since 34 is ASCII code for quotation mark))
- `"A[22]"` (the string "`A[22]`")

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

b. Produce the simplest DFA you can for these literals. Just to keep things simple when labeling transitions, feel free to use the abbreviations 'C' to mean "blank or any graphic character other than quotation mark or left square bracket; 'H' to mean "any hexadecimal digit"; and '$\bar{H}$' to mean "anything other than a hexadecimal digit."

c. Produce the simplest NFA you can for these literals. It should be simpler (have fewer states or arrows) than the DFA in part (b).

2. [10 points] In the following, do not worry about syntax trees or semantic actions; just consider the language being recognized. Symbols that don't appear to the left of some ':' are terminals. It should *not* be necessary to build LALR(1) machines to answer any of these questions!

a. I ran the following through Bison (`Prog` is the start symbol):

```
Prog :  Stmts ;
Stmts :  /* EMPTY */ | Stmt Stmts ;
Stmt :  DO Stmts UNTIL Expr ';' ;
Stmt :  UNTIL Expr DO Stmts END ';' ;
Stmt :  ID '=' Expr ;
Expr :  Term Expr2 ;
Expr2 :  /* EMPTY */ | '+' Expr ;
Term :  ID Arg  ;
Term :  '(' Expr ')' ;
Arg :  /* EMPTY */ | '(' Expr ')'  ;
```

The parser generator told me that I had shift/reduce conflicts involving UNTIL. In your answer, show why this is by including an input that illustrates the problem, and show the parse stack at the point it occurs. Does the problem occur because the grammar is ambiguous?

b. The following grammar has a couple of problems when one tries to build an LL(1) parser for it. Symbols that don't ever appear on the left of '→' are terminals, and `S` is the start symbol.

```
S → Cst Ex
Cst → '(' Ty ')' | ϵ
Ty → INT | Arr
Ex → Te Ex2
Ex2 → '-' Ex | ϵ
Te → Fa Te2
Te2 → '*' Te | ϵ
Fa → ID | '(' Ex ')'
Arr → Ty '[' ']'
```

Specifically, two of the LL(1) table entries appear to have conflicts. For each conflict, give an example of an input that runs into the error. Are the conflicts caused by ambiguity?

c. In the same grammar as for part b, above (and ignoring the problems you found there), I add some Bison-like semantic actions:

```
Ex2 → ϵ               { $$ = null; }
      | '-' Ex        { $$ = $2; }
Ex → Te Ex2           { if ($2 == null)
                            $$ = $1;
                        else
                            $$ = new SubtractNode ($1, $2);
                      }
```

This probably is not what we really want. Why not?

d. Consider the following Pyth (or Python) program fragment:

```
x = 3
z = 13

def f (y):
   x = 0
   while y > 0:
       prev_x = x
       x *= z
       y -= 1
   return x     # <<<

def g (x, y, z):
   if x > y:
       return f (z)
   else:
       return f (-z)

g (2, 5, 11)
```

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

e. For the same program as in part (c), would the behavior of the class change at all if Pyth were dynamically scoped? If so, how?

3. [1 point] At a distance of 4×10^8 kilometers from its sun, a certain planet is traveling at 50 km/sec. Its orbit is quite eccentric. How fast is it traveling when it is 10^8 kilometers from its sun?

4. [9 points] Java has a rule that one must not use the value of a variable unless it has first been "definitely assigned," meaning informally that it is clear to the compiler that the variable must have been assigned to in any execution. Normally, this is a matter for the semantic analyzer, but in absurdly simple languages, the parser may be able to do it. Consider the following grammar:

```
prog :  stmts                              { ________________________________ }

stmts :  /* EMPTY */                       { ________________________________ }

stmts :  stmts stmt                        { ________________________________ }

stmt :  ID '=' expr ';'                    { ________________________________ }

stmt :  IF expr THEN stmts ELSE stmts FI ';'

                                           { ________________________________ }

expr :  ID                                 { ________________________________ }

expr :  NUM                                { ________________________________ }
```

I am interested in finding out whether a specific variable, 'x', must always be assigned to before it is used. It is assigned to when it appears to the left of the '='. It is used in any `expr` that contains the `ID` 'x'. After an if-statement, it is definitely assigned to iff it is definitely assigned to in both the then-statements and the else-statements (ignoring the test). Questions start on the next page. Here are some examples of good and bad programs:

`OK:`	`a = b;`
	`if a then`
	`   x = a;`
	`else`
	`   x = 3;`
	`fi`
	`b = x;`
		`a = b;`
		`x = c;`
			`if a then`
			`   b = c;`
			`else`
			`fi`

`Errors:`	`a = b;`
	`if a then`
	`   x = a;`
	`else`
	`   c = a;`
	`fi`
	`b = x;`
		`a = b;`
		`c = x;`
			`if x then`
			`   x = 3;`
			`else`
			`   x = 4;`
			`fi`

a. Fill in the semantic actions for the grammar above so as to assign a 2-tuple of boolean values (u, a) as the semantic value of each non-terminal, where u is true iff 'x' is used anywhere in the text matched by that non-terminal before it is assigned to in that text, and a is true iff 'x' is definitely assigned to in the text matched by that non-terminal. The semantic actions should also check that any use of 'x' happens only after it has been definitely assigned to; at least one of the actions should report an error otherwise by calling `ERR()`. Just use Pyth notation for creating tuples and fetching the values in them.

b. Fill in the *recursive-descent compiler* on the next page so that the input program gets checked for syntactic correctness and the right errors get reported. 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 errors. As in lecture, 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  NUM  IF  THEN  ELSE  FI  ';'  '⊣'
   ```

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

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

   `ERR ()` reports an error.

   If you couldn't get part (a), you can still get full credit on this part by showing where each action in part (a) is supposed to fit in your solution to this part.

   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 introduce new types. If needed, however, you can introduce more functions (do mention somewhere what they parse). Although this skeleton suggests using Pyth for the actions, we're actually not interested in the syntax you use for writing these functions, as long as your intent is clear.

```
def prog ():
```

```
def stmts ():
```

```
def stmt ():
```

```
def expr ():
```

5. [8 points] A certain language has the following terminal symbols :

```
X : * @ < -|
```

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

```
X : < X * @ -|
```

and performs the following actions:

1. Shift `X`.
2. Reduce 0 symbols (from the stack), producing (the non-terminal symbol) `c`.
3. Reduce 2 symbols, producing `b`.
4. Reduce 1 symbol, producing `a`.
5. Shift `:`, then $<$, then `X`, and then `*`.
6. Reduce 1 symbol, producing `c`.
7. Reduce 2 symbols, producing `b`.
8. Reduce 1 symbol, producing `a`.
9. Shift `@`.
10. Reduce 0 symbols, producing `c`.
11. Reduce 4 symbols, producing `b`.
12. Reduce 3 symbols, producing `a`.
13. Shift `-|`, accepting the string.

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 `r`n (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 `r`n entries we've given you. **Important:** *there is more than one correct answer to this question!*

	Action/Goto								
State	`X`	`:`	`*`	`@`	`<`	`-|`	*a*	*b*	*c*
0.									
1.									
2.									
3.									
4.									
5.									
6.									
7.									
8.									
9.									
10.									
11.									
12.									

r1 ____________	r5 ____________
r2 ____________	r6 ____________
r3 ____________	r7 ____________
r4 ____________	r8 ____________