Final

Name:

General Instructions

You have 120 minutes to complete this exam. You may put your solutions on the test, or create additional files. You may refer to your notes, coursework, textbook, and the Internet. You may use your computer to write code, but I do not expect you to compile and execute what you write. You can, if you want, but your solutions do not need to be that polished. A good solution demonstrates that you understand the basic concepts. You may neither observe nor consult your classmates, or anyone else. Read the problems carefully. They are not in any particular order. If you have a question, ask during the test:

Zoom: 9634 792 6575
Email: `buff@cs.boisestate.edu`

Please submit your exam by 7:15 PM, with the command:

```
onyx$ submit jbuffenb cs354 final
```

I have provided the grammar from Interpreter Assignment #2, on the last page of this exam. It will be useful, for some of the problems.

Problem	*Description*	*Points Awarded*	*Points Possible*
1	Grammars: Parsing		5
2	Translation: Stack Frames		10
3	Languages: Declarative and Functional		10
4	Languages: Names		10
	Total		35

Name:

Problem #1

5

According to the grammar from Interpreter Assignment #2, the following program can be parsed in two ways, as indicated by the indentation:

```
if ··· then
    if ··· then
        ···
    else
        ···
```

(left)

```
if ··· then
    if ··· then
        ···
else
    ···
```

(right)

Show a parse tree for the program on the right, containing "···" for subtrees.

Name:

Problem #2

10

Suppose the following C program is just about to execute the indicated `return` statement. Draw a picture of a reasonable upward-growing *static-link-based* run-time stack. Show all of the stack frames, including: links, return addresses and values, formal parameters and values, local variables and values, and the stack and frame pointer. You may need to make some assumptions. You may use the next page.

```
#include <stdio.h>

void f(int i) {
  int x,y;

  int g(int i) {
    int x;
    x=i+1;              // x=3,5
    y=x+1;              // y=4,6
    if (y==4)
      return g(y);      // recursion: g(4)
    else
      return y+1;       // return 7, YOU ARE HERE
  }

  x=i+1;                // x=1
  y=x+1;                // y=2
  printf("%d\n",g(y)); // g(2)
}

int main() {
  f(0);
}
```

Name:

Name:

Problem #3

10

This problem has six (little) parts, continued on the next page. Be sure to answer all the parts, and identify which is which.

While trying to implement a common computation, the following Prolog clauses came out of your fingertips:

```
foo([]).
foo([_]).
foo([X | [Y | T]]) :- X =< Y, foo([Y | T]).
```

1) In English, what does it mean? Give a sentence.

2) In Scheme, what does it mean? Give a function definition.

Name:

3) What common computational problem were you likely trying to solve? Give one word.

4) Does `foo` solve that problem? Why or why not?

5) Would the following rule help? Why or why not?

```
foo([X | [Y | T]]) :- foo([Y | [X | T]]).
```

6) Does `foo` even *help* solve that problem? What else would you need?

Name:

Problem #4
10

The language of Interpreter Assignment #2 allows a program to use variables. However, as with AWK, variables are not declared before use. Using a variable, for the first time, implicitly declares it. This has advantages and disadvantages.

Explain what would need to be changed, in a solution to Interpreter Assignment #2, to require that a variable be declared before use. Be thorough, and explain clearly, with examples (as needed).

Name:

This is the grammar from Interpreter Assignment #2:

```
prog     : block
block    : stmt ';' block
         | stmt
stmt     : assn
         | 'rd' id
         | 'wr' expr
         | 'if' boolexpr 'then' stmt
         | 'if' boolexpr 'then' stmt 'else' stmt
         | 'while' boolexpr 'do' stmt
         | 'begin' block 'end'
assn     : id '=' expr
expr     : term addop expr
         | term
term     : fact mulop term
         | fact
fact     : id
         | num
         | '(' expr ')'
         | '-' fact
boolexpr : expr relop expr
addop    : '+'
         | '-'
mulop    : '*'
         | '/'
relop    : '<'
         | '<='
         | '>'
         | '>='
         | '<>'
         | '=='
```