CS 423 Final Examination **Name ______________________________**

Answer all questions. This is an open book, open note (take-home) exam.

1. (10 points) The regular expression for identifiers, `[a-zA-Z_][a-zA-Z_0-9]+` shows how one regular expression can flexibly catch a wide range of words (variable names). Suppose a language defined a token for a similar open-ended set of operators, controlled by some regular expression like [+\-*/%!=^&]+ that accepts lexemes like ++++ or += or -%-. What extra considerations would be needed in order to use such a token in an otherwise pretty normal expression grammar that supports traditional infix binary and prefix unary operators like `x + y` or `sqrt(x*x)` or `-z` for example?

2. (10 points) In many programming languages, reserved words would be legal variable names... if they were not reserved words. In a typical Flex specification, how is it that a reserved word such as `while` is not handled by the regular expression for identifiers, `[a-zA-Z_][a-zA-Z_0-9]+`

3. (20 points) Write a regular expression that will match proper names of people – space-separated sequences of one or more names consisting of initial capital letters followed by lowercase letters for the 2nd and subsequent characters. No hyphens, but if the next-to-the-last name is “al” or “de” it does not need to be capitalized, as in “Juan de Fuca” or “Tom al Sharif”.

4. (10 points) Near the start of this semester, students were given a Bison grammar that was directly based on a classic ANSI C grammar from the original Kernighan and Ritchie book, “The C Programming Language” that is understood to be the language definition. Why was that grammar not that great for our project? What changes had to be made to that grammar in order for us to use it?

5. (10 points) Consider a recursive context free grammar rule in Bison, such as

E : E + E ;

Whether it is left-recursive, or right-recursive, or internally- or indirectly recursive, what property or properties must be established for each recursive grammar rule, in order for them to be usable?

6. (20 points) Two alternatives for populating symbol tables are: (1) during parsing, or (2) after parsing, during a tree traversal. Describe the pros and cons of each of these alternatives. Which is more difficult, and why? Use pseudo-code or a diagram to describe how one of these alternatives might be implemented.

7. (20 points) A compiler for a more robust language such as C allows new local scopes for each compound statement block (surrounded by curly braces). In such a language, how many local scopes deep must the compiler be prepared to handle in doing symbol table lookups? Describe how you would organize your symbol tables in order to support such a feature. Would such local scopes have any impact on the number of temporary variables required, or their allocation and management?

8. (30 points) Describe the use of symbol tables as it relates to the implementation of structs. Now suppose you had to implement "C++ structs", which are structs with member functions invoked with the syntax x.*f*(*params*) where x is the struct, f is the member function name, and params are the declared parameters for that function. The semantics of calling x.f(…) are that in addition to the declared parameters, x itself is also passed into f() as as extra parameter, and its fields are visible within the function body of f. Suppose also that C++ structs provide the option of declaring fields or functions to be private where they can only be accessed inside such a function. What might you do in your symbol tables in order to support such C++ struct types and their functions?

9. (20 points) Analyze the C code fragment below. Draw syntax trees for the executable statement(s). Report what a compiler's type checker would do in order to determine whether the types were correct. Then report what the outcome of type checking would be.

```
#include <stdio.h>
int jingle();
int main()
{
   int j;
   scanf("%d", &j);
   printf("%d\n",  j + jingle);
}
```

10. (20 points) Draw a syntax tree for the following C executable statement(s). Generate intermediate three address code (in the form of a linked list diagram) for them.

```
#include <stdio.h>
int main()
{
   int i=2, j=3, k;
   scanf("%d", &k);
   k = (k + i) * j / i;
}
```

11. (25 points) Write a pseudo-code skeleton that outlines the basic control structure of an intermediate code generator. In addition to the main functionality, what auxiliary tasks or helper functions will be needed? What is the data structure constructed by an intermediate code generator? How is that data structure created?

12. (20 points) What are the primary tasks involved in final code generation? Write a pseudo-code skeleton that outlines the basic control structure of a final code generator.

13. (30 points) What information is needed in order for a code generator to generate the labels for all the conditional and unconditional branch instructions in the output code? How is that information computed and where is it stored?

14. (30 points) In classic, non-short circuit languages such as Pascal, code generation for Boolean expressions can be handled in the same manner as for arithmetic expressions over numbers. C, C++ and many modern languages, however, define a short circuit evaluation semantics that requires additional considerations. Give an example boolean expression in which the outcome would be different if short-circuit evaluation is used than the outcome if the boolean expressions are fully evaluated. Then sketch out the three-address intermediate code for the short-circuit evaluation of your boolean expression.