

CSE-504 Compiler Design

Spring '12

Final Exam

May 9, 2012

Duration: 2 hour, 30 minutes.

Name: _____

USB ID Number: _____

INSTRUCTIONS

Read the following carefully before answering any question.

- Make sure you have filled in your name and USB ID number in the space above.
- Write your answers in the space provided; Keep your answers brief and precise.
- The exam consists of **7** questions, in **13** pages (including this page) for a total of **30** points.

GOOD LUCK!

Question	Max.	Score
1.	3	
2.	3	
3.	5	
4.	5	
5.	5	
6.	5	
7.	4	
Total:	30	

1. [3 points] Consider a filter that reads text files and outputs only XML-style comments that are in that file. An XML-style comment begins with `<!` and ends with `>`. Write a Lex specification, with no C/C++ code, that implements such a filter.

Example:

Input	Output
<pre><! First line> <file> This file is an example input for the filter program. <! Contains many comments> The actual contents are unimportant; note <! And some more comments> how the comments may be interleaved with text <tag>and how they look in the output</tag> </file> <! Last line></pre>	<pre><! First line><! Contains many comments><! And some more comments><! Last line></pre>

If you give a Lex specification with C/C++ actions for this problem, you will receive partial credit.

2. [**3 points**] Consider the following grammar describing floating point numbers written in binary notation (with F as the start symbol):

$$F \longrightarrow N . N$$

$$N \longrightarrow N B$$

$$N \longrightarrow B$$

$$B \longrightarrow 0$$

$$B \longrightarrow 1$$

Write a **Yacc/Bison specification** of the above grammar, and use that specification to compute the *value* of a binary floating point number. For example the value of 101010.101 is 42.625.

For full credit, your solution should compute attributes using only side-effect-free actions.

3. [5 points] A grammar is said to be ϵ -free if either it has no ϵ -productions, or the only ϵ -production is $S \rightarrow \epsilon$ where S is the start symbol.

A grammar is said to be in Chomsky Normal Form (CNF) if it is (i) ϵ -free, (ii) its start symbol S does not appear on the right hand side of any production, and (iii) every production is of the form $A \rightarrow BC$ where B and C are some non-terminal symbols, or of the form $A \rightarrow a$ where a is some terminal symbol.

Let G_2 be a grammar in CNF that is LL(1) as well as SLR(1). Let G_2 have t terminal symbols, n non-terminal symbols and p productions.

- (a) [1 point] What is the size of the LL(1) parsing table for G_2 ? Why?

- (b) [1 point] What is the size of the SLR(1) parsing table for G_2 ? Why?

- (c) [1 point] Let s be a string of length k in $\mathcal{L}(G_2)$, i.e. the language of G_2 . What is the size of the parse tree for s (i.e. number of nodes in the tree)? Justify.

(d) [1 point] Let s be a string of length k in $\mathcal{L}(G_2)$, i.e. the language of G_2 . How long does it take to parse s using the LL(1) parsing algorithm? Justify.

(e) [1 point] Let s be a string of length k in $\mathcal{L}(G_2)$, i.e. the language of G_2 . How long does it take to parse s using the shift-reduce parsing algorithm using the SLR(1) action and goto tables? Justify.

4. [5 points]

Consider the following Decaf program:

```
class A {
    public int i;
    public int m(int i) {return i;}
}
class B extends A {
    public float m;
    public float i(float x) {return x;}
}
class C extends B {
    public float m(float x) {return m;}
    public int i(int i) {return i(i);}
}
class D extends C {
    float i;
    int m;
    void f() {
        A a; B b; C c; D d;
        a = new A();
        b = new B();
        c = new C();
        d = new D();

        /**/ EXAMPLE_EXPRESSION /**/
    }
}
```

Fill the following table such that the table lists the type of each given expression when it occurs as `EXAMPLE_EXPRESSION` in the program at the place marked with `/**/`. If there is some type error in the expression, write “Error” in that space.

	EXAMPLE_EXPRESSION	Type
1.	<code>a.i</code>	
2.	<code>b.i</code>	
3.	<code>this.i</code>	
4.	<code>super.i</code>	
5.	<code>this.i(this.m)</code>	
6.	<code>this.m(this.i)</code>	
7.	<code>this.m(this.i+this.m(this.i))</code>	
8.	<code>c.m(this.i)</code>	
9.	<code>c.m(b.i)</code>	
10.	<code>b.m(c.i(c.m))</code>	

5. [5 points] Consider adding a poor-man's version of Java's for-each loop to **Decaf**. (Aside: For-each loops are used to iterate over collections (objects that implement the collections interface) in Java, but we'll do something simpler in **Decaf**).

For-each loop is written as `for ( x : c ) { Loop Body }` where `x` is an variable and `c` is an array whose elements are the same type as `x`. The array `c` may be explicitly specified, or may be computed by a more complex expression (e.g. return value of a method). The loop works by assigning `x` in each iteration to successive elements of `c`, and then executing the *Loop Body* statement. The loop terminates when all elements of `c` are exhausted.

For-each loop	Traditional for loop
<pre> A ai; /* A is some class */ A ac[]; for (ai : ac) { ... Loop Body ... } </pre>	<pre> A ai; /* A is some class */ A ac[]; int i; for (i=0; i< ac.size; i++) { ai = ac[i]; ... Loop Body ... } </pre>

- (a) [1 point] Describe the representation of For-Each loop in the Abstract Syntax Tree.

- (b) [2 points] Write syntax-directed definition for type checking For-Each loop.

- (c) [2 points] Write syntax-directed definition to generate code for For-Each loops. You may generate CREAM code (as in HW5) or three address code.

For code generation purposes, you may assume that CREAM has an instruction `asize` that, given an array handle on top of stack, replaces it with the size (number of elements) of that array.

If you are generating three-address code, you may assume that there is an instruction `asize` such that `asize t1, a` will move array `a`'s size (number of elements) into `t1`.

6. [5 points] Consider the following fragment of 3-address instructions.

```
(1)  i=2
(2)  if (i>n) goto (7)
(3)  t1 = 4 * i
(4)  a[t1] = 1
(5)  i = i + 1
(6)  goto (2)
(7)  k = 0
(8)  i = 2
(9)  if (i>s) goto (22)
(10) t2 = 4 * i
(11) t3 = a[t2]
(12) if (t3==0) goto (20)
(13) k = k + 1
(14) j = 2 * i
(15) if (j > n) goto (20)
(16) t4 = 4 * j
(17) a[t4] = 0
(18) j = j + i
(19) goto (15)
(20) i = i + 1
(21) goto (9)
(22) count = k
```

(a) [1 point] Draw the control flow graph of this fragment, with basic blocks as the nodes of this graph.

- (b) [2 points] Identify common subexpressions in the intermediate code fragment and eliminate repeated computations of these common subexpressions.

- (c) [2 points] Identify induction variables in the intermediate code fragment perform strength reduction based on the identified induction variables.

7. [4 points] Consider the following fragment of three-address code:

```
(1)  t1 = 1
(2)  t2 = 2
(3)  t3 = t1 + t2
(4)  t3 = t2 + t3
(5)  t4 = t2 + t3
(6)  t4 = t4 + t2
(7)  t5 = t4 + t3
(8)  t6 = t5 + t3
(9)  t6 = t6 + t4
```

(a) [2 points] Let **a**, **n**, **s** and **count** be live at the end of instruction (9) in the intermediate code fragment. Compute the set of live variables at the *beginning* of the following instructions:

Instruction	Live Variables
(9)	
(8)	
(5)	
(3)	

- (b) [2 points] Draw the register interference graph for the variables in the intermediate code fragment. Based on the graph, determine the smallest number of registers needed to execute the intermediate code fragment assuming all data is kept in registers.