

## CS143 Practice Final

---

### Material

The final will cover material from all lectures, but will focus a bit more on the second half (semantic analysis, code generation, implementation issues). Material similar to the homework will be emphasized, although you are responsible for all topics presented in lecture and in the handouts (although the material on optimization will be omitted from the final—I promise.) Here's an impressive list of everything you've learned this quarter.

- Overview of a compiler – terminology, general task breakdown
- Regular expressions – writing and understanding regular expressions, equivalence between regular expressions/NFA/DFA, conversion among forms (Thompson's rules, subset construction)
- Lexical analysis – scanner generators, lex/flex
- Grammars – Chomsky's hierarchy, parse trees, derivations, ambiguity, writing simple grammars
- Top-down parsing – LL(1) grammars, grammar conditioning (removing ambiguity, left-recursion, left-factoring), first and follow set computation, top-down recursive descent implementation, table-driven predictive parsing
- Shift-reduce parsing – building LR(0) and LR(1) configurating sets, parse tables, tracing parser operation, shift/reduce and reduce/reduce conflicts, differences between LR, SLR, and LALR
- Comparisons between parsing techniques – advantages/disadvantages: grammar restrictions, parser code maintenance, future flexibility, space/time efficiency, error-handling, etc.
- Syntax-directed translation – attribute grammars, inherited and synthesized attributes, use of yacc attributes and actions, simple error-handling
- Type systems – named/structural equivalency, type compatibility and subtyping, strong/weak typing, static/dynamic typing, generics/polymorphism
- Scoping – variable lifetime and visibility, static/dynamic scoping
- Object-oriented issues – type compatibility, polymorphism, inheritance, static/dynamic binding, runtime implementation of objects and classes
- Intermediate representations – abstract syntax trees, TAC, intermediate code generation
- Target machine – processor architectures, memory hierarchies, data representation, instruction sets, executable format, assemblers, linkers, and loaders.

- Runtime issues – address space layout, function call protocols, parameter-passing, runtime stack, dynamic memory management, register use
- Language design issues – tradeoffs between efficiency, safety, and expressiveness, CT versus RT decision-making

### Problem 1

A C comment begins with `/*` and ends with the first subsequent `*/`. For each of the following regular expressions, indicate whether it exactly recognizes the language of C comments. If incorrect, give an example of a string in the language that is rejected or one not in the language that is accepted. To avoid confusion with the Kleene closure, we used `s` for a literal `*` in our regular expressions.

regular expression	correct	example string that fails & why
<code>/s.*s/</code>		
<code>/s([<sup>s</sup>/]?s)+/</code>		
<code>/s[<sup>s</sup>] *s+ ([<sup>/s</sup>] [<sup>s</sup>] *s+)* /</code>		
<code>/s([<sup>s</sup>]   s[<sup>/</sup>]) *s/</code>		
<code>/s(s? ([<sup>s</sup>]   [<sup>/</sup>])) *s/</code>		

## Problem 2

For each grammar below, answer yes or no to each of the three questions. Obviously, we don't expect you to go through the entire parser construction to answer these.

Is this grammar	LL(1)?	LR(1)?	Unambiguous?
...			
$S \rightarrow aSa \mid aSb \mid c$			
$S \rightarrow Sa \mid bS \mid c$			
$S \rightarrow aSa \mid bS \mid c$			
$S \rightarrow aSa \mid \emptyset$			
$S \rightarrow Sa \mid b \mid c$			

## Problem 3

In Java, the list of type qualifiers includes additions such as `public static final synchronized` and `transient`. Any or all of those qualifiers can appear in a declaration, in any order, but none can be repeated. Consider an unambiguous grammar for Java type qualifiers. If Java required that the options appear in a particular order would this simplify or complicate the grammar? Explain. Briefly describe how such a language feature would typically be handled in a compiler (i.e. how could a compiler allow all subsets in any order without repetition and disallow all others).

## Problem 4

The following input file was used to generate a bison parser. Its associated scanner just returns each character read as an individual token.

```
%{
  int num = 0;
}%
%left '&'
%%
S : S '&' S {$$ = $1 + $3; printf("Yow %d\n", $$);}
  | L
  | /* empty */ {$$ = num = 100;}
  ;

L : E {$$ = $1 * 2; printf("Hi\n");} L {printf("Bye\n"); $$ = $1 * $2;}
  | E {$$ = $1 * 10; printf("Ack\n");}
  ;

E : 'A' {$$ = ++num;}
  ;
```

Show the output printed from the parser when reading the input `AA&&A`

**Problem 5**

- Give an example of an inherited attribute used in parsing Decaf programs.
- Inherited attributes are easier to incorporate into a top-down parser than a bottom-up one. Briefly explain why this is so.
- Describe a technique that can be used to support inherited attributes in bison.

**Problem 6**

Java allows methods to be overloaded. The Java overloading resolution rule is: when given a choice of multiple methods in a class with the same name, the method with the most specific matching argument types is selected. (It is a compile-time error if there is no unique method with the most specific argument types.) For example, consider `a.act(b)` where `a` has type `Binky`. The `Binky` class has two methods `act(Object x)` and `act(Winky x)`. If `b` is of class `Dinky`, where `Dinky` is a subtype of `Winky`, then the overloading resolves to the `act(Winky x)` method.

Java resolves overloading at compile time using the static types, while dynamic dispatch selects methods to execute at run time using the dynamic types. This is all as usual, but overloading has tricky interactions with dynamic dispatch. Consider the following program skeleton:

```
// All access restrictions (public/protected/private) omitted for clarity
class Point{
    boolean equal(Point x){...} // Version 1
}
class ColorPoint extends Point{
    boolean equal(ColorPoint x){...} // Version 2
}

Point p1 = new Point();
Point p2 = new ColorPoint();
ColorPoint cp = new ColorPoint();
```

The overloading in this example is in the subclass `ColorPoint`, which has two `equal` methods (the one in the class and the inherited one from `Point`).

For each method call listed below, indicate which version of `equal` is called.

Indicate version 1 or 2

<code>p1.equal(p1);</code>	
<code>p1.equal(p2);</code>	
<code>p1.equal(cp);</code>	
<code>p2.equal(p1);</code>	
<code>p2.equal(p2);</code>	

<code>p2.equal(cp);</code>	
<code>cp.equal(p1);</code>	
<code>cp.equal(p2);</code>	
<code>cp.equal(cp);</code>	

### Problem 7

Pascal compilers are required by the Pascal Standard to implement `var` parameters by reference. Describe an experiment to detect a non-compliant compiler that is implementing `var` parameters by value-result instead of by reference. You may use any reasonable language syntax for your example.

### Problem 8

Assuming you are dealing with a compiler for a Decaf-like language. For each of the following errors, circle which phase would normally issue the message: (L) lexer, (P) parser, (SA) semantic analyzer, (CL) code generator/linker, (R) runtime.

Misspelled built-in (e.g. <code>Print</code> )	Circle one:	L	P	SA	CL	R
Array index out of bounds		L	P	SA	CL	R
<code>break</code> appears outside a function		L	P	SA	CL	R
Unknown method called on object		L	P	SA	CL	R
One-d array accessed with two subscripts		L	P	SA	CL	R
Identifier name that begins with an underscore		L	P	SA	CL	R
<code>length()</code> called on null array		L	P	SA	CL	R
Function declared and used but not defined		L	P	SA	CL	R
Wrong number of arguments in function call		L	P	SA	CL	R
Comma inside number (e.g. 1,000)		L	P	SA	CL	R

### Problem 9

Imagine that some Decaf class declares a method but doesn't define it, and that the PP4 specification requires it be reported as a link error. For this question, we will instead handle pure virtual methods (those that are declared and not defined) like Java and C++, treating the class as abstract, intended for use only as a base class and disallow creating an actual instance. Describe the specific changes would you need to make to your compiler correctly support this behavior. Be sure to think about all phases of compilation. Some issues to consider: how do you detect that a class is abstract? What effect does it have on subclassing? Is it still valid to declare of variable of an abstract type? What constitutes semantically valid use of an abstract class? When is it legal to call a pure virtual method? Do you need to emit the code for methods in a abstract class? Do you need to emit its

vtable? Will this change the opportunities for optimization? What effects does it have on runtime behavior?

### Problem 10

You are to add the C-style ternary as a new expression to Decaf. This expression is like a compact if-then-else. It evaluates the test and if true, evaluates the first expression, otherwise it evaluates the second. Only one of the two expressions is actually evaluated at runtime.

`a > b ? a : b`

- What changes will you need to make to the standard Decaf scanner to support the new operation? Show your changes by providing code snippets that are added, removed, or changed in the lex input file.
- Assume the ternary expression was added to the parser without establishing its associativity or precedence. Give an example input that would result in a parser conflict or ambiguity due to the lack of precedence. Describe the conflict.
- What changes will you need to make to the standard Decaf parser to support this operation? Be specific in terms of tokens, types, precedence, and productions that are added, removed, or changed in the yacc input file. The ternary expression should have the lowest precedence of all operators.
- What changes will you need to make to the standard Decaf semantic analyzer to support this operation? Be specific in terms of what validity checks and error messages must be added, removed, or changed. Describe what action is taken on each of your productions from part c). You need not write out the actual code, a clear description is sufficient. Do not add any global variables.
- Add the necessary calls to the `codeGenerator` class needed to generate correct code for the ternary operation. Clearly indicate how the actions are associated with your productions from the third bullet above. Show the actual code for this task. Do not add any global variables.