

## CS143 Practice Final Solution

---

### Solution 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>	N	matches <code>/* */ abc /* */</code>
<code>/s([<sup>s</sup>/]?)s+/</code>	N	rejects <code>/* / */</code>
<code>/s[<sup>s</sup>] *s+ ([<sup>/s</sup>] [<sup>s</sup>] *s+)* /</code>	Y	
<code>/s([<sup>s</sup>]   s[<sup>/</sup>]) *s/</code>	N	rejects <code>/* abc **/</code>
<code>/s(s? ([<sup>s</sup>]   [<sup>/</sup>])) *s/</code>	N	<code>/* */ abc /* */</code>

### Solution 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?
$S \rightarrow aSa \mid aSb \mid c$	N	Y	Y
$S \rightarrow Sa \mid bS \mid c$	N	N	N
$S \rightarrow aSa \mid bS \mid c$	Y	Y	Y
$S \rightarrow aSa \mid \emptyset$	N	N	Y
$S \rightarrow Sa \mid b \mid c$	N	Y	Y

### Solution 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).

If you try to construct a grammar for all subsets without repetition allowing any order, the grammar will balloon outrageously because of the combinatorics. Imposing an order makes it much simpler, since you have just a sequence of options, each expands either to the qualifier or null. Languages like C and Java that have to support the former (i.e. any order) don't try to use the parser for this. The grammar is written to allow any number of the options to be specified (including duplicates) and the non-repetition constraint is handled as part of semantic analysis.

### Solution 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`

```
Hi
Ack
Bye
Yow 102
Ack
Yow 1112
```

### Solution 5

- Give an example of an inherited attribute used in parsing Decaf programs.

There are many examples to choose from: the current loop context, scoping information, the class currently being defined, etc.

- Inherited attributes are easier to incorporate into a top-down parser than a bottom-up one. Briefly explain why this is so.

Inherited attributes are those attributes being passed down from the parent to the children. Given a top-down parser constructs the tree top-down, it is a simple matter to pass along those attributes.

- Describe a technique that can be used to support inherited attributes in bison.

In a bottom-up parser, the parent is only built after the children have been recognized and assembled on the parse stack, which makes it hard to pass the attributes along since the parent doesn't yet exist. That doesn't mean it is impossible, but you do have to rely on some other mechanism such as use of global variables, embedded actions, and/or reaching back into the parse stack.

### Solution 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>	1
<code>p1.equal(p2);</code>	1
<code>p1.equal(cp);</code>	1
<code>p2.equal(p1);</code>	1
<code>p2.equal(p2);</code>	1
<code>p2.equal(cp);</code>	1
<code>cp.equal(p1);</code>	1
<code>cp.equal(p2);</code>	1
<code>cp.equal(cp);</code>	2

### Solution 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.

```
# Example is pseudo-Pascal
PROCEDURE Test(var x:integer, var y:integer)
BEGIN
  x := x + 1;
  y := y + 1;
END

VAR a: integer;
  a := 10;
  Test(a, a);
  Writeln(a); /// outputs 12 if pass by ref, 11 if value-result
```

### Solution 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. `Print`)

L, or SA

Array index out of bounds

R

break appears outside a function	P	
Unknown method called on object		SA
One-d array accessed with two subscripts		SA
Identifier name that begins with an underscore	L	
length() called on null array		R
Function declared and used but not defined		CL
Wrong number of arguments in function call	SA	CL
Comma inside number (e.g. 1,000)	P, or	SA

### Solution 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?

There are three things you would need to change: at end of parsing class declaration, examine class scope for any declared but not defined methods, if any found, mark class as abstract. In semantic analysis, disallow calls to New for an abstract class. For code generation, do not emit the vtable for abstract classes. Everything else stays the same. You can still declare variables of abstract class type. Subclasses can inherit from an abstract class. If they don't implement the required methods, they, too will be abstract. Semantic analysis for object use, method send, etc. is as before. Code is generated normally for all concrete methods. There is no advantage for optimization, dynamic dispatch is still required.

## Solution 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.

Add `?` and `:` to the character set for single-character operators.

- 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.

`a > b ? a : b` can be parsed as either `a > (b ? a : b)` or `(a > b) ? a : b`. There is a shift-reduce conflict in the parser after reading `a > b`. Do we reduce the `a > b` already on the stack or shift the `?` to keep building? That's the conflict that would need to be resolved.

- 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.

Before all other precedence directives add:

```
%left '?' ':'
```

To the productions for Expr add

```
Expr '?' Expr ':' Expr
```

A good instinct is to make the first term BoolExpr instead of Expr but that will create reduce/reduce conflicts. The reason for this is a little obscure and we accepted BoolExpr as full credit anyway.

- 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.

ReportError if \$1 does not have boolType (suppress if already errorType)

ReportError if \$3 and \$5 do not have equivalent (or compatible types)  
(suppress if one or both is errorType)

Set \$\$ to be more general of \$3/\$5 type or errorType if types are not compatible

- 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.

```
Expr  {$<label>$ = cg->NewLabel(); cg->GenIfZ($1, $<label>);}
'?'
Expr  {$<decl>$ = cg->GenTempVar($4->GetType());
      cg->GenAssign($<decl>$, $4);}
':'  {$<label>$ = cg->NewLabel(); cg->GenGoto($<label>);}
      cg->GenLabel($<label>2);}
Expr  {$$ = $<decl>5; cg->GenAssign($$, $8); cg->GenLabel($<label>7);}
```