CIS (4|5)61
Spring 2015 Final Exam key

In addition to programming languages, we use grammars and parsers for notations used to save, restore, and transmit data. Translating between an internal memory format and an external notation is commonly known as *serialization* and *deserialization*. Current popular formats for serialization include Javascript object notation (JSON), XML, Java serialization format, and Python pickles.

The following two questions are based loosely on a simplified version of JSON. You don't need to know JSON to solve the problem.

(score)

1. [10 points]

Consider the following grammar for trivial object notation (TON) where the token v is used to represent an atomic value like an integer or quoted string. (The literals "`{`", "`:`", etc. are also tokens.)

Ton → **Obj** *$*

Obj → "{" **Items** "}"

Items → **Items Item** | /* empty */

Item → **Pair** | **Value**

Pair → **Key** "`:`" **Value**

Key → **Value**

Value → v

This grammar is sufficient to represent simple objects like

```
{ "key1": 99  88 "key2": 13 },
```

but not nested objects like

```
{  "key1": { 44  88 { 13 22 } }  "key2": 13 }.
```

Make the simplest modification to the grammar you can with the following properties:

- Your modified grammar should be LALR(1)
- Your modified grammar should allow arbitrarily nested values, but
- Your grammar must not allow nested objects as keys. For example, this is not a valid object:

```
{  {"k": 44}: 99 }.
```

We can add Obj as an alternative for Item. I believe that's the simplest possible change that allows nested objects.

Ton → **Obj** *$*

Obj → "{" **Items** "}"

Items → **Items Item** | /* empty */

Item → **Pair** | **Value** | **Obj**

Pair → **Key** "`:`" **Value**

Key → **Value**

Value → v

(score)

2. [10 points] Show that your modified grammar is still LALR(1). (You can draw just enough of the CFSM to show that your change does not introduce ambiguity.)

3. [5 points] When we try to produce an LALR(1) parser, we might find reduce-reduce conflicts, and we might find shift-reduce conflicts, but there is no such thing as a shift-shift conflict. Why?

4. [5 points] The original FORTRAN IV compilers did not allocate local variables in stack frames. Rather, each variable had its own fixed address in memory. Also, FORTRAN IV did not support recursion. Could these two facts about FORTRAN be related? How?

5. [5 points] Recall that a graph-coloring register allocator uses an interference graph. There is an edge in the interference graph between variable v and variable w if v and w have overlapping live ranges. Why do overlapping live ranges determine whether the same register can be assigned to v and w?

(score)

6. [10 points] It's annoying to have to declare the type of every variable in Java. It's annoying when my Python program crashes because of some simple mistake that a type checker would have caught. It seems like we have to choose between annoying type declarations and annoying (or dangerous) crashes, but maybe not. *Type inference* is increasingly used to reduce the need for declarations while still having strong, compile-time type checking.

Consider the type-checking rule for 'let' in Cool, which I have simplified by omitting the special case for SELF_TYPE.

$$[\text{LET}]\ \frac{\begin{array}{l} O, M, C \vdash e_1 : T_1 \\ T_1 \leq T_0 \\ O[T_0/x], M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \texttt{let}\ \ x : T_0 \leftarrow e_1\ \ \texttt{in}\ \ e_2 : T_2}$$

In Cool the initialization expression e_2 is optional, so the type declaration is really needed (otherwise we wouldn't know what the default initialization should be). But let's turn it around: Let's require the initialization and get rid of the type declaration entirely, giving the variable the type of the initialization expression. Modify the type inference rule above to do that, so for example

```
let x <-  o.foo(y) in x.bar()
```

becomes legal in Cool, giving x the same type as `o.foo(y)`. (You can write the new type inference rule or just indicate changes to the rule above. Please write clearly.)

(score)

7. [10 points] In programming languages that do not require variable declarations, simple typographical errors like misspelling a variable can cause run-time errors. For example, this Python function has an infinite loop caused by misspelling 'low' as 'law':

```
def badary_search(ar, key):
    low = 0
    high = len(ar) - 1
    while low <= high:
        mid = (high + low) // 2
        cur = ar[mid]
        if cur > key:
            high = mid - 1
        elif cur < key:
            law = mid + 1
        else:
            # They must be equal
            return mid
    return -1
```

Can you propose a data flow analysis to catch this error and others like it at compile time? It can be application of a standard data flow analysis property, or a novel one that can easily be implemented as a bit-vector analysis. (A simpler check for identifiers that appear only once may work for this case, but I'd like a data flow analysis that would work even if the same misspelling occurs more than once.)

8. [10 points] A common optimizing transformation is to reuse a value rather than recomputing it. This is called "common sub-expression elimination." For example, if we had

```
x <-  y*3 + 5;
z <-  y*3 - 7;
```

the compiler might produce code that computes `y*3` just once and uses it twice. The programmer could also make the same transformation in this example, but there are other opportunities for reuse that only the compiler can apply. For example, consider this Cool code for adding three items to a list:

```
l.add(1);
l.add(2);
l.add(3);
```

What opportunity do you see for common sub-expression elimination in this code? (Consider what you know about the representation of objects and classes in Cool, and how dynamic dispatch works.)