3F6

SOFTWARE ENGINEERING AND DESIGN

Engineering Tripos Part IIA

2009 Solutions

Steve Young

Question 1

(a) The parse trees of the two examples are shown in Fig. 1.

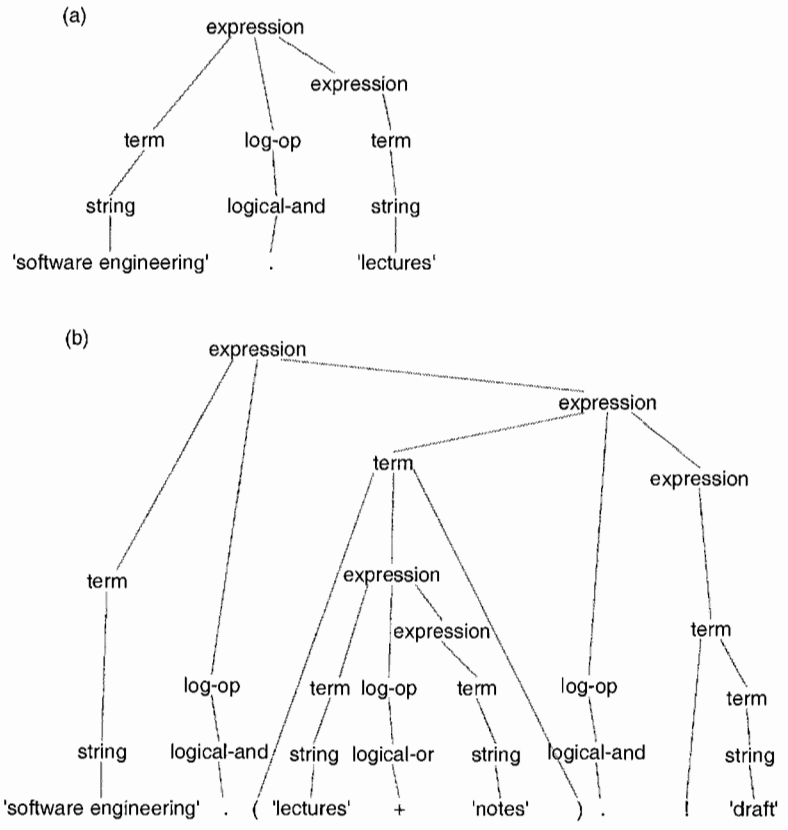

Figure 1:

(b) The UML class diagram is shown in Fig. 2. Note that there is no need to derive classes for the operators since they can be encoded in the node that dominates them. In all cases, the document arg of `IsaMatch` is assumed given.

(c) The implementation of the `IsaMatch` methods are also shown in Fig. 2.

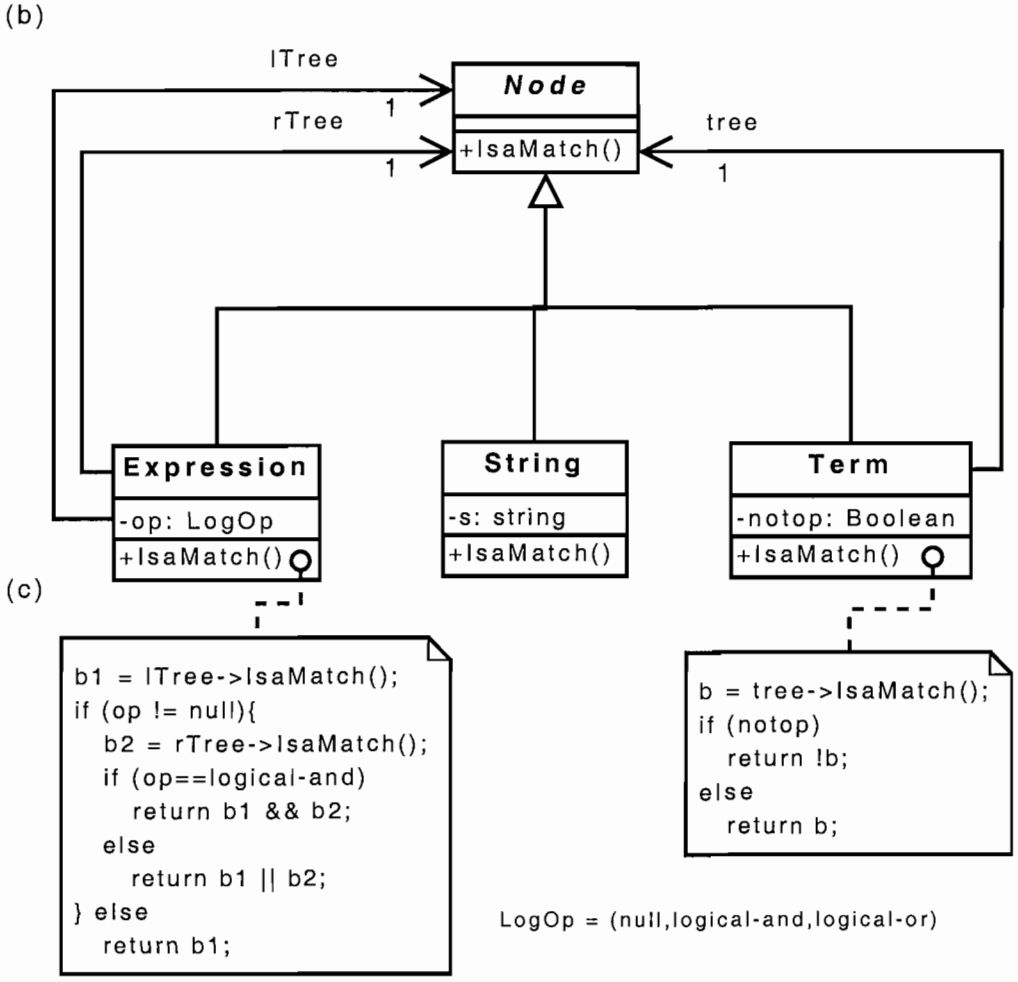

Figure 2:

(d) To elevate the precedence of "." over "+", a new non-terminal node is required to dominate the "." operator. Use the existing *term* node for this, and introduce a new node called *factor* to serve the previous term role:

```
expression => term [logical-or expression]
term       => factor [logical-and term ]
factor     => string | "(" expression ")" |  notop factor
```

In the UML diagram, terms would be renamed factors and a new term class added similar to expressions. The op attribute can be removed since expressions now dominate only the "+" operator and terms only dominate the "." operator. The code for `IsaMatch()` would simplify accordingly.

Question 2

(a) The observer pattern allows multiple observers to view some system or state called the subject such that any change to the subject state is immediately notified to each of the observers. The pattern works by simply requiring each new observer to register itself with the one or more subjects. Whenever the subject state changes, its Notify method is called to signal the change to all the registered observers.

- Advantages: decouples the subject from the observers allowing the subject code to be developed independently of the observers; the subjects do not need to know who is observing them
- Disadvantages: an observer can cause a change in a subject unaware that the change might invoke a long sequence of possibly complex updates in other observers.

(b) The sequence diagram is shown in Fig. 3

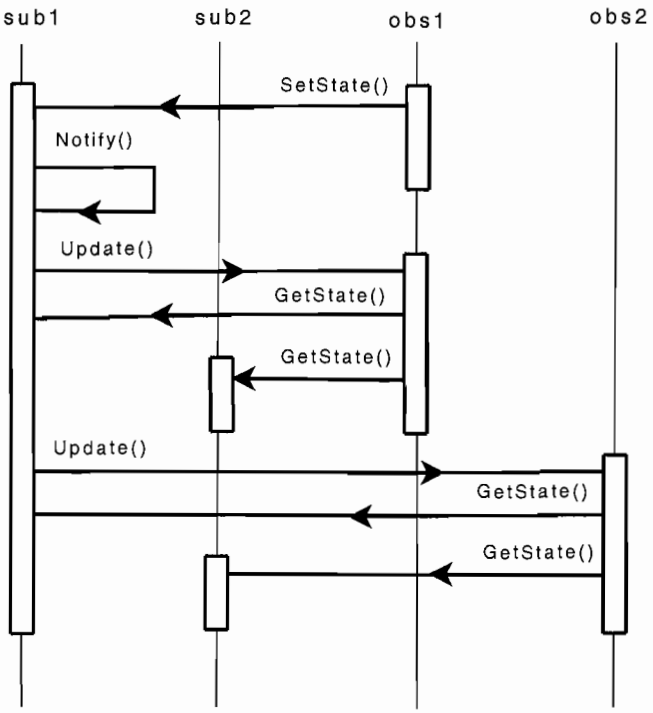

Figure 3:

(c) If the subjects are independent then observers only need to know which specific subject is requesting an update. This can be achieved by passing a reference to each subject in the Update() call as shown in Fig. 4

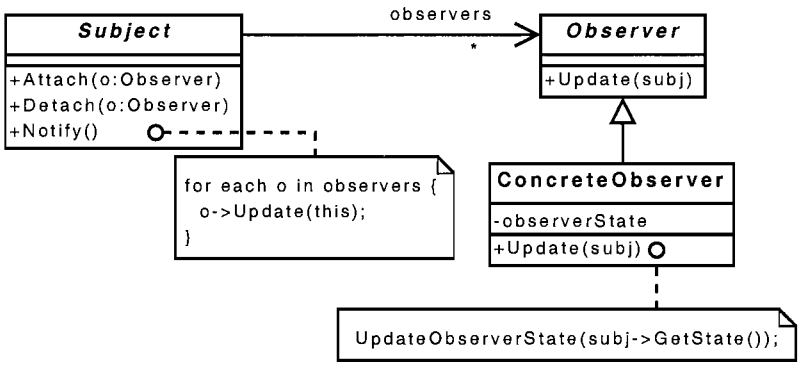

Figure 4:

(d) The two basic requirements are to store a list of observers with each subject, and the ability to mark each observer as "seen". Using the STL template library, a possible solution would then be as follows.

```
// define a type to represent a list of observers
typedef list<Observer *> ObserverList;
// now declare the subject -> observer list mapping
map<Subject *,ObserverList> somap;
// and the seen flags
map<Observer *,bool> seen;
// the global list of observers shown in UML diagram (Fig 3)
ObserverList observers;
```

The required methods are then

```
void Register(Subject *s, Observer *o){
   if (o not in observers) add o to observers;
   if (o not in somap[s]) add o to somap[s];
}
void Notify(Subject *s){
   for (each o in somap[s]) seen[o] = true;
}
void NotifyComplete(){
   for (each o in observers)
      if (seen[o]) {
         o->Update();  seen[o] = false;
      }
}
```

Note that we assume that each observer remembers the subjects that it has attached itself to. A more complex solution might send a list of just those subjects which have

been updated to the observer, but this is not required in this case (as indicated by the lack of arguments in the call to `Update()` in Fig 3).

Whilst the above uses associative maps, simple arrays with appropriate lookup and insert operations would be equally acceptable.

Question 3

(a) The term *Thread-safe* when applied to a set of functions (eg a class or a library) means that the functions can be called simultaneously by multiple parallel threads in such a way that the functionality is maintained. When non-thread-safe routines are called, wrong values may be returned and/or the system may crash as a result of invalid memory references. The major cause of non-thread-safeness in libraries is the use of global variables. Where a global is essential, then it must usually be protected by a semaphore.

(b) A semaphore is a gating device to ensure that a critical section of code is only executed by at most one thread a time. It is implemented within the operating system using a flag variable and a list of thread context records representing all of the threads waiting to access this semaphore. The operation is as follows:

```
Secure():
   if (flag set){
      put calling thread on waiting list;
      resume next thread in ready queue
   }
Release():
   if (waiting list empty)
      unset flag;
   else
      move top of waiting list to ready queue;
```

where the *ready queue* is the list of threads maintained by the operating system which are ready for execution.

(c) Signals allow a thread to voluntarily suspend itself waiting for a signal until another thread sends that signal. They are often used with semaphores to allow a thread to wait inside a critical section until some resource becomes available. In this case, when a thread suspends itself its hold on the semaphore is released, however, once it is signalled there is an implicit call to resecure the semaphore before it can proceed.

(d) Semantics:

When a thread calls `GetKey` it returns an object of type `ResourceKey`. If there is no key available, the calling thread is suspended until one is available.

When a thread calls `ReturnKey` it passes an object of type `ResourceKey` back to the key pool. The key passed must have been acquired by a previous call to `GetKey`. The call always returns immediately and is never blocked.

```
ProtectedKeyPool::GetKey(){
   poolAccess.Secure();
   while (keys->KeysAvailable()==0){
      keyAvail.Wait();   // wait til a key is available
   }
   key = keys->GetKey();
   poolAccess.Release();
   return key;
}
ProtectedKeyPool::ReturnKey(key){
   poolAccess.Secure();
   keys->ReturnKey(key);
   keyAvail.Send();     // signal any waiting threads
   poolAccess.Release();
}
```

(e) To provide priority access, the simplest solution is to use two signals, say, `hiKeyAvail` and `loKeyAvail`. When a low priority thread needs to wait for a key, it calls `loKeyAvail.Wait()` otherwise it calls `hiKeyAvail.Wait()`. In addition, a count is made of all threads waiting in the high priority queue. In `ReturnKey`, the count is checked and the appropriate signal is sent ie

```
ProtectedKeyPool::GetKey(bool hi){
   poolAccess.Secure();
   while (keys->KeysAvailable()==0){
      if (hi) {
         ++hicount;  hiKeyAvail.Wait(); --hicount;
      } else
         loKeyAvail.Wait();
   }
   key = keys->GetKey();  poolAccess.Release();
   return key;
}
ProtectedKeyPool::ReturnKey(key){
   poolAccess.Secure(); keys->ReturnKey(key);
   if (hicount>0)
      hiKeyAvail.Send();
   else
```

```
        loKeyAvail.Send();
    poolAccess.Release();
}
```

Note that in the exam, just the verbal description of the modification would have been sufficient.

Question 4

(a) **Atomicity** each transaction operation must be atomic, otherwise it would be impossible to recover cleanly from an aborted transaction.

Consistency transactions must leave the system in a consistent state. This is essential since any sequence of transactions can be aborted at any point.

Isolation results of an incomplete transaction must not be visible to any other transaction. Otherwise, another transaction might see an inconsistent state and produce erroneous results.

Durability the system must not fail between a transaction committing and the results of the transaction being recorded in the system state.

(b) The resource allocation graph is shown below in Fig. 5 upto step 12 (`A.read`)at which point the system is dead-locked.

(c) The corresponding wait-for-graph at the point of deadlock is shown below in Fig. 6.

(d) Recovery from deadlock involves

- aborting a transaction which will break the deadlock loop (called the victim)
- rolling back to the last checkpoint
- redoing the remaining transactions
- restarting the victim at some later point in time

The victim is typically chosen according to the following criteria

- has not been running for a long time
- has made few updates
- is blocking multiple transactions

In this case, the choice lies between T1 and T3 since aborting neither T2 nor T4 will clear the deadlock. Hence, T3 is chosen on the grounds of recency and fewer updates.

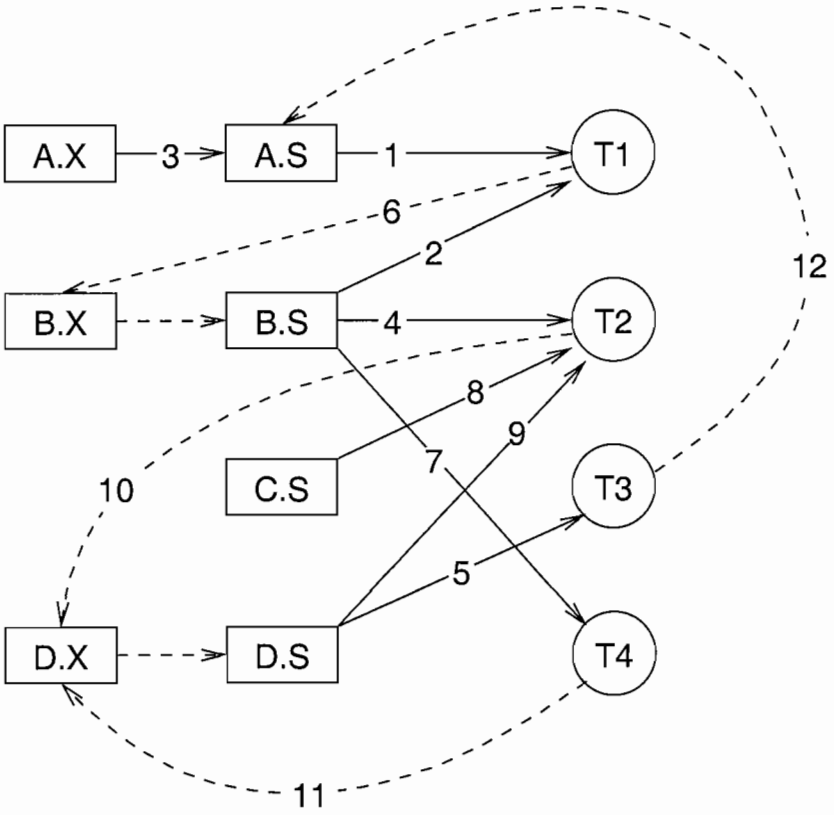

Figure 5:

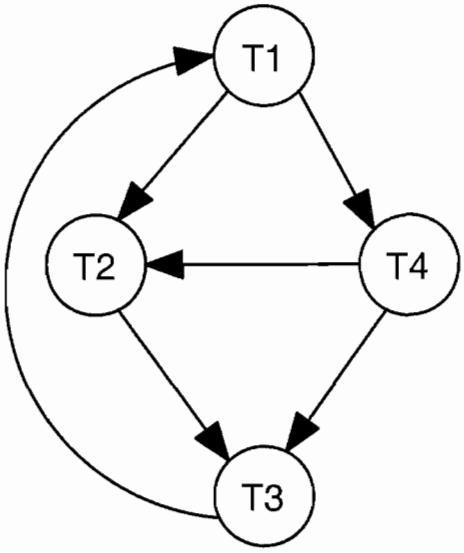

Figure 6: