1. In two sentences, explain the difference between the following terms. Give an example to illustrate the difference.

   a. **Static vs. dynamic storage allocation**: With static storage allocation a compiler can allocate storage to data knowing only the text of the program; with dynamic storage allocation, the decision can only be made at run time. Global variables are statically allocated; local variables in recursive procedures are dynamically allocated on the run-time stack.

   b. **Parameters vs. local variables of a procedure**: Parameters are variables used to transmit values to a called procedure from a calling procedure. Local variables are variables within a procedure whose scope is the procedure itself; they are used to store values during the execution of the procedure. For example, in the procedure `foo` below, `n` is a parameter, `i` is a local variable.

   ```c
   void foo(int n) { // n is a parameter
      int i;         // i is a local variable
      ...
   }
   ```

   c. **Call graph vs. flow graph**: A call graph is a directed graph in which there is a node for each procedure and call site (place in the program where a procedure is invoked), and an edge from call site `c` to procedure `p`, if `c` calls `p`. A flow graph is a directed graph in which there is a node for each basic block in the program and an edge from block `B` to block `C` if it is possible for the first instruction in `C` to immediately follow that last instruction in `B`. In the procedure `fib` in problem (3) there are two call sites for `fib` in the last return statement. In a flow graph, each of these call sites would be node with an edge directed towards the node for procedure `fib`. A flow graph for `fib` is shown in the answer for 3(b).

   d. **Applicative-order evaluation vs. normal-order evaluation**: In applicative-order evaluation, all the parameters of a procedure are evaluated before the procedure is called. In normal-order evaluation, the procedure is called and then the parameters are evaluated when and if they are needed. C uses applicative-order evaluation for functions and normal-order evaluation for macros.

   e. **Type-safe language vs. strongly typed language**: There is no standard definition for a type-safe language. Vijay Saraswat has perhaps the best
definition: A language is type safe if the only operations that can be performed on data are the ones sanctioned by the type of the data. A strongly typed language, on the other hand, is one in which type errors are always detected either statically (at compile time) or dynamically (at run time). ML is generally regarded to be a type-safe language. Because of unions as well as other reasons, C is not strongly typed.

2. Data layout.

b. Allocation of local variables in an activation record: Here the local variables can be allocated on the stack. We assume the stack is growing downward and we have aligned each local variable to occupy 4 bytes.

```c
void foo() {
    char c;
    int a[10];
    float f;
}
```
b. Allocation of a variable-sized array: Here we store a fixed-sized pointer within the activation record to the variable-sized array whose elements are stored on top of the activation record.

```c
void foo(int n) {
    char c;
    int a[n];
    float f;
}
```

3. Consider the C function:

```c
int fib(n) {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return (fib(n-1) + fib(n-2));
}
```
a) Three-address code:

```plaintext
fib: if n == 0 goto L0
    if n == 1 goto L1
    t1 = n - 1
    param t1
    t2 = call fib, 1
    t3 = n - 2
    param t3
    t4 = call fib, 1
    t5 = t2 + t4
    return t5
L0:  return 0
L1:  return 1
```

b. Flow graph:

```
B1  if n == 0 goto B4
    if n == 1 goto B5
    t1 = n - 1
    param t1
    t2 = call fib, 1
    t3 = n - 2
    param t3
    t4 = call fib, 1
    t4 = t2 + t4
    return t5
B4  return 0
B5  return 1
```
4. Consider the C function above.

a) Activation tree for \texttt{fib(3)}:

```
  fib(3)
      /   \
    /     \nfib(2)   fib(1)
      /   \  /
    /     / \
fib(1)  fib(0)  
```

b) Activation records on stack when \texttt{fib(2)} is invoked for the first time. Here we assume \texttt{fib(3)} has called \texttt{fib(2)}. We assume the stack is growing upward and the frame pointer for each AR points to the actual parameter.

```
AR for fib(2)

AR for fib(3)

AR for main

---

<table>
<thead>
<tr>
<th>return value = ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 2</td>
</tr>
<tr>
<td>control link</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>return value = ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 3</td>
</tr>
<tr>
<td>control link</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>return address</th>
</tr>
</thead>
</table>
```

5
5. Using Algorithm 9.17, compute the sets of expressions available at the entry and exit of each block of the flow graph of Fig. 9.10.

\[ U, \text{ the universal set of expressions, is} \]

\[ \{ a+b, c-a, b+d, e+1, b*d, a-d \} \]

We will use a bit-vector of length 6 to represent a subset of \( U \).

\[
\begin{array}{|c|c|c|}
\hline
\text{Block} & \text{\( e_{\text{gen}} \)} & \text{\( e_{\text{kill}} \)} \\
\hline
\text{B1} & 000 000 & 111 011 \\
\text{B2} & 110 000 & 011 011 \\
\text{B3} & 000 000 & 001 011 \\
\text{B4} & 100 000 & 001 111 \\
\text{B5} & 010 000 & 101 110 \\
\text{B6} & 000 001 & 111 011 \\
\hline
\end{array}
\]

Applying Algorithm 9.17, the values of \( \text{IN}[B] \) and \( \text{OUT}[B] \) after each iteration are as follows:

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Block} & \text{\( \text{OUT}[B]^0 \)} & \text{\( \text{IN}[B]^1 \)} & \text{\( \text{OUT}[B]^1 \)} \\
\hline
\text{B1} & 111 111 & 000 000 & 000 000 \\
\text{B2} & 111 111 & 000 000 & 110 000 \\
\text{B3} & 111 111 & 110 000 & 110 000 \\
\text{B4} & 111 111 & 110 000 & 110 000 \\
\text{B5} & 111 111 & 010 000 & 010 000 \\
\text{B6} & 111 111 & 010 000 & 000 001 \\
\text{EXIT} & 111 111 & 000 001 & 000 001 \\
\hline
\end{array}
\]

The values of \( \text{IN}[B] \) and \( \text{OUT}[B] \) remain the same at the second iteration, so the algorithm terminates after two iterations.

6. Prove by induction that in Algorithm 9.17, the \( \text{IN} \) and \( \text{OUT} \) sets never grow.

We will prove by induction on \( n \), the number of iterations of the inner for-loop in Algorithm 9.17, that for \( n \geq 1 \),

\[ \text{IN}[B]^{n+1} \text{ is a subset of } \text{IN}[B]^n \text{ and } \text{OUT}[B]^{n+1} \text{ is a subset of } \text{OUT}[B]^n. \]

(If \( B \) is a subset of \( A \), then \( B \) can be equal to \( A \).)

**Basis step:** Initially, \( \text{OUT}[\text{ENTRY}]^0 \) is set to the empty set and \( \text{OUT}[B]^0 \) is set to \( U \), the set of all expressions of the form \( x + y \) where \( x + y \) is the right side of some three-address statement, for all blocks \( B \) other than \( \text{ENTRY} \).
The first iteration sets $\text{IN}[B] = \bigcap_{\text{predecessor of } B} \text{OUT}[P]$ and $\text{OUT}[B] = e_{\text{gen}}_B \cup (\text{IN}[B] - e_{\text{kill}}_B)$.

**Inductive step:** Suppose that $\text{IN}[B]^n$ is a subset of $\text{IN}[B]^{n-1}$ and $\text{OUT}[B]^n$ is a subset of $\text{OUT}[B]^{n-1}$. Now consider what happens in the $n$th iteration. $\text{IN}[B]^n$ is set to the intersection of $\text{OUT}[P]^{n-1}$ for all predecessors $P$ of $B$. By the inductive hypothesis, we can assume no $\text{OUT}[P]$ has grown from the previous iteration, so the value of $\text{IN}[B]^n$ must be a subset of the value of $\text{IN}[B]^{n-1}$.

Similarly in the $n$th iteration $\text{OUT}[B]^n$ is set to $e_{\text{gen}}_B \cup (\text{IN}[B] - e_{\text{kill}}_B)$. The values of $e_{\text{gen}}_B$ and $e_{\text{kill}}_B$ remain the same for each iteration. By the inductive hypothesis we can assume no $\text{IN}[B]$ has grown from the previous iteration so the value of $\text{OUT}[B]^n$ must be a subset of the value of $\text{OUT}[B]^{n-1}$.

We can now conclude that after each iteration $\text{IN}[B]$ is a subset of its previous value and $\text{OUT}[B]$ is a subset of its previous value.