1. Here's the list:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Kind</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>procedure</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>variable</td>
<td>INTEGER</td>
</tr>
<tr>
<td>sub1</td>
<td>procedure</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>type</td>
<td>(= STRING)</td>
</tr>
<tr>
<td>y</td>
<td>variable</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>sub2</td>
<td>procedure</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>variable</td>
<td>t (= STRING)</td>
</tr>
<tr>
<td>b</td>
<td>variable</td>
<td>t (= STRING)</td>
</tr>
<tr>
<td>z</td>
<td>variable</td>
<td>t (= STRING)</td>
</tr>
</tbody>
</table>

2. (a) Here are two leftmost derivations for the same sentence:

\[
E \Rightarrow E \text{ or } E \Rightarrow id \text{ or } E \Rightarrow id \text{ or } E \Rightarrow id \text{ or id and id} \\
E \Rightarrow E \text{ and } E \Rightarrow E \text{ or } E \Rightarrow id \text{ or id and id}
\]

(b) Here's a suitably rewritten grammar:

\[
E \rightarrow E \text{ or } T \\
E \rightarrow T \\
T \rightarrow T \text{ and } F \\
T \rightarrow F \\
F \rightarrow \text{not } F \\
F \rightarrow (E) \\
F \rightarrow \text{true} \\
F \rightarrow \text{false} \\
F \rightarrow id
\]

This problem is completely analogous to arithmetic expressions. Note that in disambiguating, I've not only enforced the given precedence order, but also made both \text{and} and \text{or} left-associative. The alternative with

\[
E \Rightarrow T \text{ or } E \\
T \Rightarrow F \text{ and } T
\]

is also an acceptable answer, since the the problem didn’t ask for a particular associativity.

3. (a). A grammar is LL(1) if and only if its predictive parsing table has no multiply-defined entries. Consider the right-hand sides of the first and third productions for \text{S}. The terminal \text{ (} is in FIRST(\text{(}) and also in FIRST(\text{(A)}). Therefore the table entry for the row labeled \text{S} and the column labeled \text{ (} will have (at least) two entries for these two productions. So the grammar cannot be LL(1). (Note that there was no need to calculate any FOLLOW() sets after all!)

(b) This requires removing left-recursion \text{and} left-factoring:

\[
S \rightarrow ( S' ) \\
S \rightarrow a \\
S' \rightarrow ) \\
S' \rightarrow A ) \\
A \rightarrowSA' \\
A' \rightarrow "SA' \\
A' \rightarrow \epsilon
\]
(c) Here's C/Java-like code:

```c
void s() {
  if (token == '(') {
    advance();
    s1();
  } else if (token == 'a')
    advance();
  else error();
}

void s1() {
  if (token == ')')
    advance();
  else {
    a();
    if (token == ')')
      advance();
    else
      error();
  }
}

void a() {
  s();
  a1();
}

void a1() {
  if (token == ',') {
    advance();
    s();
    a1();
  }
}
```

(d) First rewrite `a1` as a `while` loop; then inline `a1` into `a`, `a` into `s1`, and finally `s1` into `s`.

```c
s() {
  if (token == '(') {
    advance();
    if (token == ')')
      advance();
    else {
      s();
      while (token == ',') {
        advance();
        s();
      }
      if (token == ')')
        advance();
      else
        error();
    }
  } else if (token == 'a')
    advance();
  else
    error();
}
```
4.(a) One answer:

```
command → filename rdin rdout
         → filename rdout rdint
         → filename rdin pipe
rdin → ’<’ filename
      → ε
rdout → ’>’ filename
      → ε
pipe → ’|’ filename pipe
      → ’|’ filename rdout
```

4b.

![Diagram of the syntax tree for commands with file input and output, pipes, and filenames.](image)
5. (a) Regular expressions for patterns:

Number \([0-9]+\)
Short \([A-Za-z]\)
Long \([A-Za-z]+\)$

(e) Example: abcde0
(Only after the 0 is read does the machine discover that it is not reading a long abcde... rather than the short a. Characters bcde0 will be rescanned on the next invocation of the lexer.)