1. Here's the list:

Identifier	Kind	Type
`main`	procedure	
`x`	variable	INTEGER
`sub1`	procedure	
`t`	type	(= `STRING`)
`y`	variable	BOOLEAN
`sub2`	procedure	
`a`	variable	`t` (= `STRING`)
`b`	variable	`t` (= `STRING`)
`z`	variable	`t` (= `STRING`)

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

$E \Rightarrow E$ `or` $E \Rightarrow$ `id or` $E \Rightarrow$ `id or` E `and` $E \Rightarrow$ `id or id and` $E \Rightarrow$ `id or id and id`
$E \Rightarrow E$ `and` $E \Rightarrow E$ `or` E `and` $E \Rightarrow$ `id or` E `and` $E \Rightarrow$ `id or id and` $E \Rightarrow$ `id or id and id`

(b) Here's a suitably rewritten grammar:

$E \rightarrow E$ `or` T
$E \rightarrow T$
$T \rightarrow T$ `and` F
$T \rightarrow F$
$F \rightarrow$ `not` F
$F \rightarrow$ `(`E`)`
$F \rightarrow$ `true`
$F \rightarrow$ `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 `and` and `or` left-associative. The alternative with

$E \rightarrow T$ `or` E
$T \rightarrow F$ `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 S. The terminal `(` is in FIRST(`()`) and also in FIRST(`(`A`)`). Therefore the table entry for the row labeled S and the column labeled `(` 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 *and* left-factoring:

$S \rightarrow$ `(` S'
$S \rightarrow$ `a`
$S' \rightarrow$ `)`
$S' \rightarrow A$ `)`
$A \rightarrow SA'$
$A' \rightarrow$ `,`SA'
$A' \rightarrow \epsilon$

(c) Here's C/Java-like code:

```
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();
  }
}
```

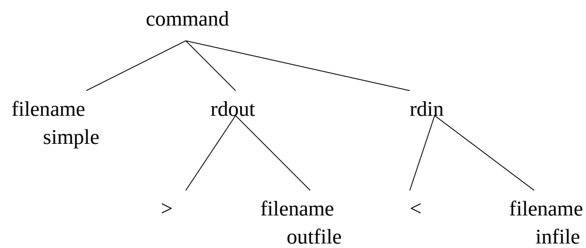
(d) First rewrite `a1` as a `while` loop; then inline `a1` into `a`, `a` into `s1`, and finally `s1` into `s`.

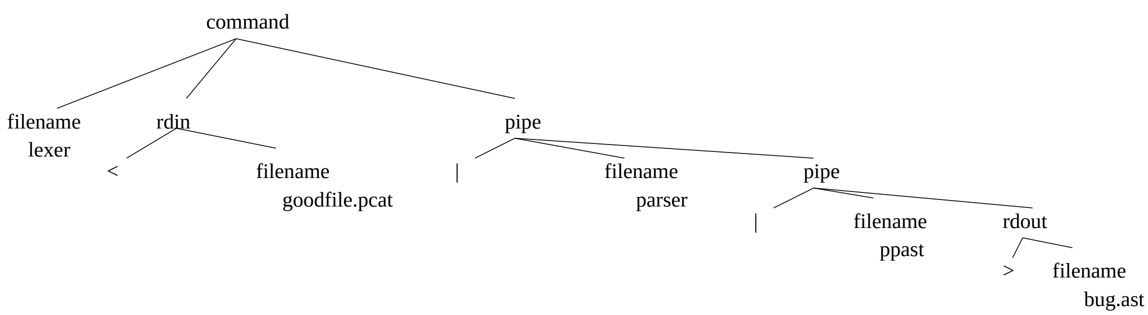
```
s()
{
  if (token == '(') {
    advance();
    if (token == ')')
      advance();
    else {
      s();
      while (token == ',') {
        advance();
        s();
      };
      if (token == ')')
        advance();
      else
        error();
    }
  } else if (token == 'a')
    advance();
  else
    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.

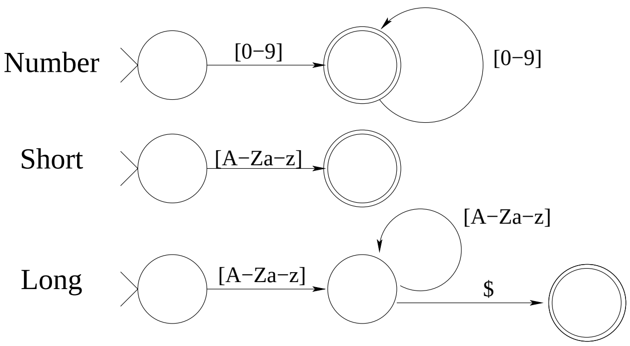




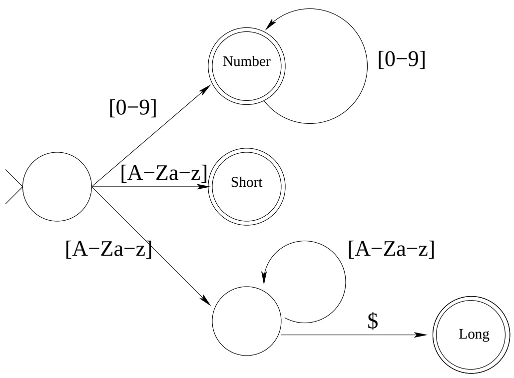
5. (a) Regular expressions for patterns:

Number `[0-9]+`
Short `[A-Za-z]`
Long `[A-Za-z]+$`

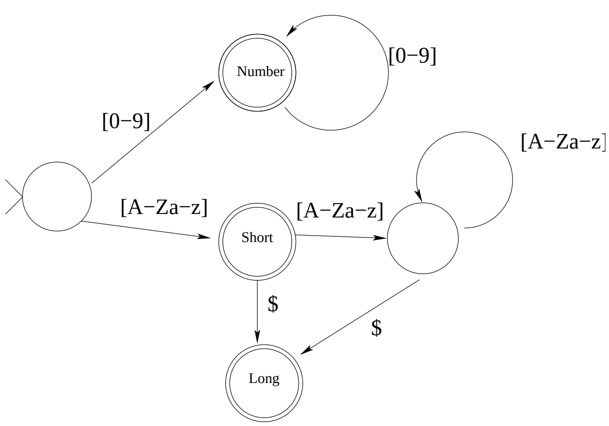
5b



5c



5d



(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.)