1 Lexical analysis

a) (2p)

Token sequences

```
ID = [a-z][a-z0-9]*
```

b) (3p)

Token sequences

```
INT = [0-9]+ ([_]* [0-9]+)*
```

or, alternatively:

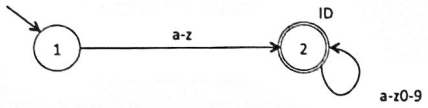
```
INT = [0-9] ([_0-9]*[0-9])?
```

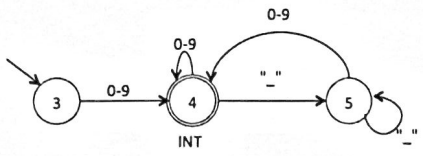
or, alternatively:

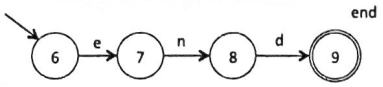
```
INT = [0-9] [_0-9]* [0-9] | [0-9]
```

c) (5p)

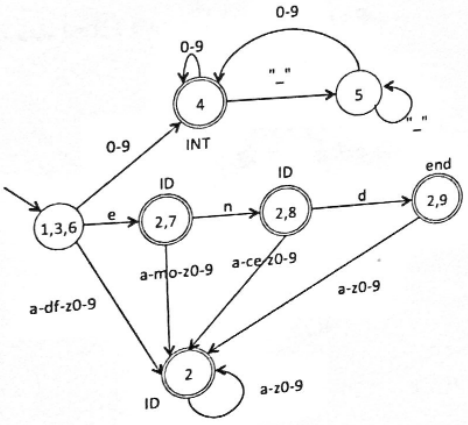
The automata:







d) (5p)

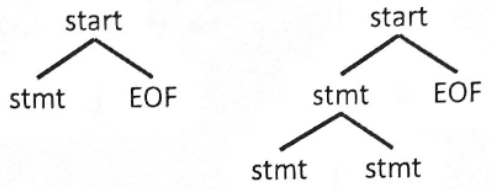

0-9
0-9
" "
4
5
" "
INT
0-9
ID
ID
end
1,3,6
e
2,7
n
2,8
d
2,9
a-mo-z0-9
a-ce-z0-9
a-df-z0-9
a-z0-9
2
a-z0-9
ID

DFA:

2 Context-Free Grammars

a) (5p)

For example, the sentence `EOF` can be derived in the following two ways:



b) (5p)

An equivalent grammar on EBNF form, with as few nonterminals and productions as possible:

```
start → stmt* EOF
stmt → "{" stmt* "}" | ID "=" NUM ";"
```

c) (5p)

Equivalent LL(1) grammar:

```
p0 : start → stmtlist EOF
p1 : stmtList → stmt stmtlist
p2 : stmtlist → ϵ
p3 : stmt → "{" stmtlist "}"
p4 : stmt → ID "=" NUM ";"
```

d) (5p)

The LL(1) table:

	"{"	"}"	ID	"="	";"	NUM	EOF
start	p0		p0				p0
stmtlist	p1	p2	p1				p2
stmt	p3		p4				

Since there is no conflict, the grammar is LL(1).

3 Program analysis

a) (5p)

Attribute grammar:

```
syn ClassDecl ClassUse.decl() = lookup(getID());
inh ClassDecl ClassUse.lookup(String s);
eq Module.getClassDecl().lookup(String s) {
  for (ClassDecl c : getClassDecls()) {
    if (c.getID().equals(s)) return c;
  }
  return null;
}
```

b) (5p)

Attribute grammar:

```
syn boolean ClassDecl.isCyclic() circular [true] {
  if (!hasSuper())
    return false;
  else if (getSuper().decl() == null) return false;
  else
    return getSuper().decl().isCyclic();
}
```

c) (5p)

Attribute grammar:

```
coll HashSet<ClassDecl> ClassDecl.subclasses() [new HashSet<ClassDecl>()] root Module;
ClassDecl contributes this
when hasSuper() && getSuper().decl() != null
to ClassDecl.subclasses()
for getSuper().decl();
```

4 Code generation and run-time systems

a) (5p)

The call `a.f(1,4,x)` is difficult to optimize because it is not known at compile time which implementation of `f` is called. The variable `a` might refer to an object of a subclass to `A` and `f` might have been overridden by another method implementation in that class.

Comments: If `f` is declared `final` the optimization can be done by the compiler. Otherwise, Java's dynamic compilation in the JVM may inline `f` at runtime, using polymorphic inline caches, and then perform the constant propagation optimization.

b) (5p)

Stack and heap: