Quiz 1

EECS665 - Compiler Construction
2019, Spring

Name: ____________________ Student ID: ______________

Do Not Open Until Instructed!

Before the Quiz starts:

- Read all of the instructions on this page
- Write your name and student ID on this page
- Retrieve your page of notes, if you have one
- Prepare your writing materials
- Put all other materials away

After the Quiz starts:

- Write your student ID (**not** your name) on all subsequent pages
- Announcements / corrections will appear on the projector
- Turn in your quiz and note page to Drew when finished.
- After the quiz time expires, answers may be presented but no new material will be given.

The quiz is worth 50 points and consists of 5 questions. You will have *35 minutes* to complete all questions. Work quickly and move on if you are stuck. If you'd like to pass the time before the quiz starts or before it ends, feel free to draw a picture of yourself in the box below:

Student ID: ____________________

NOTE: Several of the questions on this quiz refer to the DotGobbler language. This is a simple language created for the purpose of this quiz. Necessary details of the language are presented in the last 2 pages of the quiz, which do not contain questions. You are free to detach these pages from the rest of the quiz.

Question 1 (10 Points)

Create tokenization patterns for the four terminal symbol types **intlit**, **id**, **assign**, and **dot** of DotGobbler.

You may express your rules as a tokenizer action table, or a sequence of flex rules, or an automaton with the ability to return tokens and spit back input characters at final states (your choice). If you use flex syntax, your actions may simply return the token type (i.e `return dot` rather than exactly specifying `TokenTypes`, etc.)

Note: Descriptions of the tokens are written at the end of this document in the Lexical Details section. You may tear the description pages off if you like.

```
[1-9][0-9]*                 { return intlit; }
gets                        {return assign;}
"="                         { return assign; }
[.]                         { return dot; }
[a-zA-Z]*,[a-zA-Z]*         { return id;}
```

Student ID: ____________________

Question 2 (10 Points)

Create an EBNF grammar that recognizes any valid *Program* in the DotGobbler language. You may name your non-terminal symbols however you choose.

Note: Descriptions of the syntax are written at the end of this document in the Syntactic Details section. You may tear the description pages off if you like.

```
Program ::= Statements | ε
Statements ::= Statement Statements
             | Statement
Statement ::= Reference assign Expression
Reference ::= id dot Reference
            | id
Expression ::= Referance | intlit
```

Question 3 (10 Points)

Create an input to a DotGobbler compiler that would pass tokenization, but would not pass parsing.

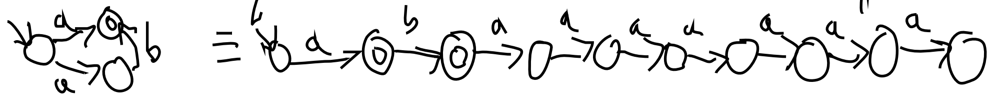
Question 4 (10 Points, Divided)

You are given an NFA with 3 states, and a claim that there is an equivalent DFA with 10 states.

Part a (5 Points):

Is it possible for such a DFA to exist? Explain your reasoning.

Yes - while an NFA with 3 states must necessarily be equivalent to ~~s~~ some DFA with $2^3 = 8$ states, another equivalent machine may have even more. Example:



Part b (5 Points)

Is it possible for a DFA with fewer states than 10 to be equivalent to the given NFA? Explain your reasoning.

Yes. The Rabin-Scott Powerset construction guarantees that any NFA with k states has an equivalent DFA with <u>at most</u> 2^k states

Student ID: ____________________

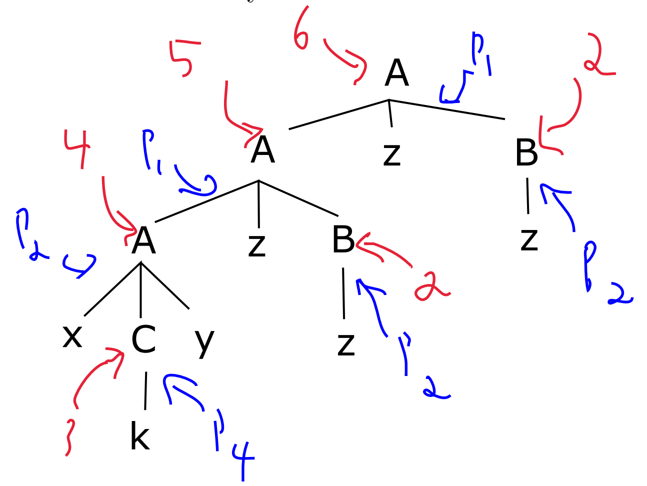
Question 5 (10 Points)

Consider the following Syntax-Directed Translation scheme:

P_1	A	::=	A z B	$A_1.trans = A_2.trans + 1$
P_2	A	::=	x C y	$A.trans = C.trans + 1$
P_3	B	::=	z	$B.trans = 2$
P_4	C	::=	k	$C.trans = 3$

(current notation) $\Rightarrow$ LHS.trans = A.trans' + 1
LHS.trans = C.trans + 1
~~LHS~~LHS.trans = 2
LHS.trans = 3

What is the syntax-directed translation for the root of the following parse tree?



$\boxed{6}$

Note that the P_3 rules need not be computed: they do not contribute to the translation of the root node

DotGobbler Lexical Details

The DotGobbler language consists of four token types:

- **intlit**: Corresponds to an integer literal. An integer literal may consist of sequences of the digits 0-9 but the first digit must not be 0.
  Examples:
  `102` is an intlit.
  `012` is not an intlit (starts with a 0).

- **id**: Corresponds to an identifier. Identifiers in DotGobbler are any sequences of alphabet characters, with two additional constraints:

  - Identifiers must include the character g at least once.
  - Identifiers must not be keywords.

  Examples:
  `anger` is a indentifier (contains a g, is not a keyword).
  `g` is a indentifier (contains a g, is not a keyword).
  `anvil` is not an identiifer (does not contain g).
  `gets` is not an identiifer (it is an assign keyword, described below).
  `getss` is a indentifier (despite containing a keyword).

- **assign**: Used for assignment. There are two sequences that match assignment

  - The word `gets`
  - The equals (=) symbol.

  Examples:
  `g gets 7` would be tokenized as **id assign intlit**.
  `ga = 28` would also be tokenized as **id assign intlit**.

- **dot**: Used for field reference. Corresponds to the period symbol (.)
  Example:
  `ga.gb.gc` would be tokenized as **id dot id dot id**.

DotGobbler Sytactic Details

- A *Program* is a sequence of zero or more *Statement*s. Note that each *Statement* is not separated by semicolons.
- A *Statement* consists of a *Reference*, followed by an assign token, followed by an *Expression*.
- A *Reference* consists of one or more **id**s separated by **dot**s. For example,
- An *Expression* consists of a *Reference* or an **intlit**.

Assume the DotGobbler tokenizer strips whitespace, including newlines. The following are valid DotGobbler programs:

Example 1

```
ga = gb
```

Example 2*: (the empty string)*

Example 3

```
ga.gb = gb
gd = ga.gb.ga
```

Example 4

```
gc = 7
gd gets 7
gm = 7
```
