CSE 401 Midterm Exam – Winter 2008 Anderson/Roberts

Name: ______Sample Solution____________________________________

CSE Email: ____

This is a "closed everything" test. Answer all questions.

Keep this page up until told to start

Total: 90 points.

Question	Max Points	Score
1	10	
2	8	
3	10	
4	16	
5	14	
6	12	
7	10	
8	10	
Total	90	

In this test the following alphabetic sets can be used.
Alpha ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
Num ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

1. [10] NOTE: **Meta-rules such as [ ] for "zero or one" are not allowed in *this* question, you may use *epsilon* if needed.**

(a) A file name base is any sequence of "properly hyphenated" letters or digits, where a sequence is properly hyphenated if it doesn't begin or end with a hyphen and there are no consecutive hyphens; e.g. **i8-a-Hot-dog**. A file name base must be at least one character long. Give a regular expression for

file_name_base ::=

No unique answer, most common was $(\text{Alpha}|\text{Num})^{+}\,(\text{-}(\text{Alpha}|\text{Num})^{+})^{*}$

(b) A filename is one or more file_name_base sequences each separated from the next by a period followed optionally by a period and an extension. An extension is exactly three letters. If there is a period in the filename then there must be an extension. So, **a** is a file name; **a.b** is not a filename, and **a.b.doc** is. Give a regular expression for

filename ::=

$\text{file\_name\_base}\ (\mathcal{E} \mid (.\text{file\_name\_base})^{*}\ .\text{Alpha Alpha Alpha})$

Most common mistakes: not allowing **base.extension**; not requiring an extension if **.file_name_base** is present.

2. [8] In the MiniJava compiler, we classify tokens into important groups. Give **two** examples for each group:

reserved word:

for, break, if, class, public, …

delimiter:

(,), {, },[,], ;, =, …

operator:

+, -, *, /, ||, …

tokens with values:

This caused the most difficulty. We were looking for IDENTIFIER, INT_LITERAL or DOUBLE_LITERAL but we gave full credit for examples of these such as 5 or 7.9E4.

3. [10] Give (a) the concrete syntax tree and (b) the abstract syntax tree for:
(c + a) * b using the grammar and MiniJava-like nodes.

E ::= E + T | T
T ::= T * F | F
F ::= id | (E)

Derivation (Concrete Syntax Tree) AST

The concrete tree did not cause too many problems, the most common mistake was not rooting the tree in E and just starting with T.

The abstract tree using MiniJava nodes turned out to be difficult; only one person got it completely correct. Common mistakes were including extra E or T nodes; labeling the operator nodes with just "+" instead of AddExpr (or MulExpr) and not including a VarExpr node for the identifiers.

4. [16] Given the following grammar: S ::= aS | aSbS | c

Please use examples or give definitions to explain your answer to the questions below.
Is this grammar:

a) Left Recursive? yes **<u>no</u>**
Why/Why not?

S does not appear as the leftmost symbol on the RHS.

b) Suitable for predictive parsing? yes **<u>no</u>**
Why/Why not?

aS is a common prefix, should be left factored

c) Ambiguous? **<u>yes</u>** no
Why/Why not?

Ex. You can generate two parse trees for aacbc

d) Regular? yes **<u>no</u>**
Why/Why not?

This has recursive structure and you cannot create a RE to recognize it (# of a's must be > # of bs')

5. [14] Given the following grammar

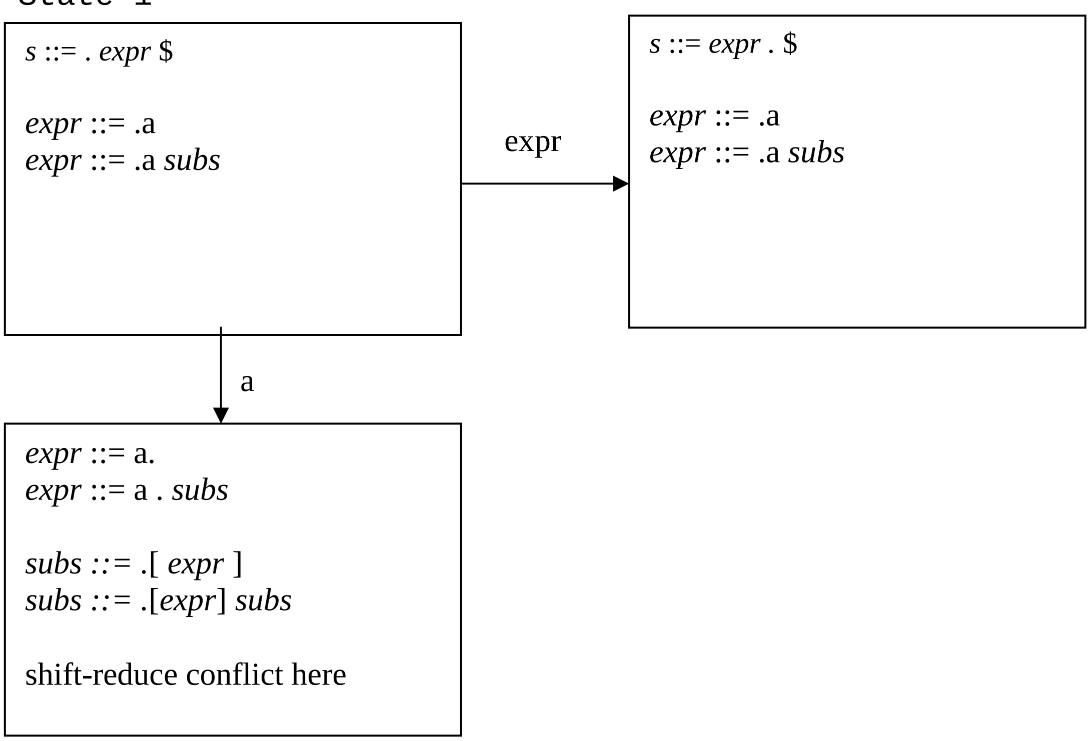
s ::= *expr* $
expr ::= a | a *subs*
subs ::= [*expr*] | [*expr*] *subs*

Build the first couple of states in the DFA for an LR parser for this grammar.

a) Form the closure for the production: *s* ::= *expr* $, shown in the box labeled `State 1` below.
b) ALSO draw and ***label the edges*** out of `State 1`.
c) ALSO show the ***complete contents (closure)of the states*** reachable by the edges drawn out of `State 1`
d) Indicate anything special about states (e.g., conflicts, reducing states)

Do not draw any edges out of other states. You should only have 2-5 states total. We are **not** asking you to draw the entire DFA.

`State 1`



6. [12] Suppose we want to add the following conditional statement to MiniJava:

```
ifequal (exp1, exp2)
  statement1
smaller
  statement2
larger
  statement3
```

The meaning of this is that *statement1* is executed if the integer expressions *exp1* and *exp2* are equal; *statement2* is executed if *exp1* < *exp2*, and *statement3* is executed if *exp1* > *exp2*. Note that `ifequal`, `smaller`, and `larger` are all keywords.

(a) [5] Give context-free grammar production(s) for the `ifequal` statement that allows either or both of the "`smaller`" and "`larger`" parts of the statement to be omitted. If both the "`smaller`" and "`larger`" parts of the statement appear, they should appear in that order. **You do not need to give productions for expressions and other types of statements, just the `ifequal` statement** (which should be considered a statement as well).

Write your grammar here:

Here are two solutions. The first one uses ε-productions

> ***stmt*** **::= ifequal (** ***exp*** **,** ***exp*** **)** ***stmt optsmaller optlarger***
> ***optsmaller*** **::= smaller** ***stmt*** **| ε**
> ***optlarger*** **::= larger** ***stmt*** **| ε**

The other one is more brute-force but doesn't include any ε-productions.

> ***stmt*** **::= ifequal (** ***exp*** **,** ***exp*** **)** ***stmt***
> **| ifequal (** ***exp*** **,** ***exp*** **)** ***stmt*** **smaller** ***stmt***
> **| ifequal (** ***exp*** **,** ***exp*** **)** ***stmt*** **larger** ***stmt***
> **| ifequal (** ***exp*** **,** ***exp*** **)** ***stmt*** **smaller** ***stmt*** **larger** ***stmt***

(b) [5] Is the grammar with your production(s) from part (a) ambiguous? If not, argue informally why not; if it is ambiguous, give an example that shows that it is.

Yes. This grammar has the same sort of problem as the "dangling else" in the usual grammar for conditional statements. There are two possible ways to derive, for example,

> **ifequal (** ***exp*** **,** ***exp*** **) ifequal (** ***exp*** **,** ***exp*** **)** ***stmt*** **smaller** ***stmt***

A derivation can be given where the "smaller" part is associated with the second "ifequal", and another can be given that associates it with the first "ifequal".

(c) [2] When compiling this statement, what rule(s) or condition(s) should the type checker verify?

Verify that expr1 and expr2 are both of type integer

7. [10] In class we discussed **static/lexical scoping** and **static typing**. What is the difference? Give a definition for both. Please give a pseudo code example if it helps your answer.

Scoping has to do with visibility of a variable and finding the correct instance of a variable to bind with.

Typing has to do with determining the type of a variable.

Static or lexical scoping determines which instance of a variable to use at compile time by examining the lexically enclosing scope (nesting seen in the code on a piece of paper)

Dynamic scoping is done at run time and uses the nesting of procedure calls to determine which variable instance to use.

Static typing is done at compile time. Dynamic typing is done at run time. You can have strong or weak typing in either case.

8. [10] Give an example to show the difference between **structural equivalence** and **name equivalence**.

Example 1: s1 and s2 are equivalent under structural equivalence (not under name equivalence)

```
struct {                          struct {
      int x;                            int x;
     } s1;                             } s2;
```

Example 2: s3 and s4 are equivalent under name equivalence (and under structural equivalence)

```
typedef struct {
 int x;
} S;
S s3;
S s4;
```

Note that this has to do with the TYPE of variables. Not equivalent values stored in variables.