# CISC 471/672 Compiler Construction
# Are you understanding Compilers?
**October 14, 2014 (Partial Exam for Studying for Fall 2015**

Name: ________________________________________ total 14 pages (2 blank)

Scores

| Question | Points for problem | Points Earned |
|---|---|---|
| A1 | 10 | |
| B1 | 5 | |
| B2 | 5 | |
| B3 | 5 | |
| B4 | 5 | |
| C1 | 10 | |
| C2 | 10 | |
| C3 | 12 | |
| C4 | 10 | |
| C5 | 13 | |
| C6 | 10 | |
| C7 | 10 | |
| Total | 100 | |

**A. General questions**

1. (10 points) True/False questions: Please answer true (T) or false (F) for the statements below. You may provide a short explanation. Marks that are unclear will be counted as wrong.

_______1. A scanner and parser communicate through the abstract syntax tree.

_______2. Code compiled on one type of machine (computer) can be run on another type of machine (computer).

_______3. Code interpreted on one machine (computer) can be interpreted on another machine as long as there is an interpreter.

_______4. Due to the complexity of programming languages today, compilers today can have many passes.

_______5. Lexical analysis tools, such as lex and flex, convert the specifications to NFA and then DFA, which then puts the results in a table.

_______6. A grammar that is ambiguous must be rewritten to be used by Bison to indicate precedence.

_______7. Bison could perform the lexical analysis rather than have Flex do it.

_______8. In SLR parsing, if A -> X. and B -> Y. are in the same state, then the grammar is not LR(0).

_______9. Left recursion in a grammar causes a top-down parser to make random choices.

_______10. All common prefixes must be removed from the grammar to create a parser with Bison.

## B. Lexical Specification and Analysis.

B1. (5 points) Draw a DFA or NFA that implements the following flex-like specification. Show the final states and which action would be performed in each final state.

| | |
|---|---|
| a+b* | {action 1;} |
| a(ba)+b | {action 2;} |
| (baa\|bab) | {action 3;} |

B2. (5 points) Consider the following flex-like specification. The alphabet is the set {a,b,c}. Parentheses are used to show the association of operations and are not part of the input alphabet.

| | |
|---|---|
| aac | {return token1;} |
| ba+c | {return token2;} |
| ba* | {return token3;} |
| (b\|c)* | {return token4;} |
| a+b | {return token5;} |
| (cac \| cab) | {return token6;} |

Show how the following string would be partitioned into tokens (by adding vertical lines) by grouping the characters into lexemes. Label each lexeme with the integer of the correct token class. Assume flex semantic for this question. Blank in the string are put there for readability – they are not part of the string.

a b c c b a a c c c a b a b a b c a c c c a a a b c a b b a a a a c b a a a

B3 (5 points). In Decaf, an identifier "is a sequence of letters, digits, and underscores, starting with a letter". Let "T_Identifier" represent the identifier token that is returned to Bison. Write a flex specification for recognizing a Decaf identifier. Note that you do not have to set the value of *yylval* in the actions. That is, replace the REGULAR_EXPRESSION_FOR_IDENTIFIER and TOKEN in the following line with correct regular expression and token:

```
REGULAR_EXPRESSION_FOR_IDENTIFIER            {return TOKEN;}
```

B4.(5 points) Given the following Bison specification for Decaf programs,

```
/* Begin of bison specification */
%union {
            Decl *decl;
            List<Decl*> *declList;
            VarDecl *var; List<VarDecl*> *varList;
            Type *type;
}
/* Non-terminal types*/
%type <declList>        DeclList
%type <var>             Variable  VarDecl
%type <varList>         VarDecls
%type <decl>            Decl
%type <type>            Type

/* Grammer rules */
%%
     Program:     DeclList            { Program *program = new Program($1);}

     DeclList:    Decl                {$$ = new List<Decl*>; $$->Append($1); };

     Decl:        VarDecl             {$$ = $1};

     VarDecl :    Variable ';'        {$$ = $1;}

     Variable :   Type T_Identifier   {$$ = new VarDecl(new Identifier(@2, $2), $1); }

/* End of bison specification */
```

Please draw the abstract syntax tree (mark each tree node with its class name) for the following program which has only one variable declaration:

```
            int a;
```

You are not required to add attributes to each tree node.

Blank page

C. **Syntax Specification and Analysis.**

C1. (10 points) Write a grammar for a very simple language that has any number (at least 1) of READ statements followed by any number (could be 0) of WRITE statements. Both READ and WRITE statements start with the keyword and surround the arguments with left and right parentheses. A read statement can only read in 1 variable at a time and a write statement can write any number of variables but at least 1.

C2. (10 points) Given the following grammar, construct the First and Follow sets and determine if the grammar is LL(1) without building the parse table.

| | | |
|---|---|---|
| Goal | → | Expr |
| Expr | → | Term Expr’ |
| Expr’ | → | + Expr’ |
| | \| | – Expr’ |
| | \| | ε |
| Term | → | Factor Term’ |
| Term’ | → | * Term |
| | \| | / Term |
| | \| | ε |
| Factor | → | num |
| | \| | id |

C3. (12 points) Construct the LL(1) table for the following grammar, given the First and Follow sets. Is the grammar LL(1)?

L $\rightarrow$ Atom
L $\rightarrow$ List
List $\rightarrow$ ( Lseq )
Lseq $\rightarrow$ L Lseq'
Lseq' $\rightarrow$ , Lseq
Lseq' $\rightarrow$ ε
Atom $\rightarrow$ num
Atom $\rightarrow$ id

| Symbol | First | Follow |
|---|---|---|
| num | num | - |
| id | id | - |
| ( | ( | - |
| ) | ) | - |
| , | , | - |
| L | num id ( | , $ ) |
| Atom | num id | , $ |
| List | ( | , $ ) |
| Lseq | num id ( | , $ ) |
| Lseq' | , ε | , $ ) |
| | | |

C4. (10 points) Consider the following grammar:

**S → Sa | Sc | c**

a. Show why the grammar cannot be parsed by a top down predictive parser.

b. Rewrite the grammar so it can be parsed by a predictive parser.

c. Using the LL(1) table, show how a table-driven parser parses the string "caca"

| | **a** | **c** | **$** |
|---|---|---|---|
| **S** | | S → c L | |
| **L** | L → aL | L → cL | L → ε |