CSE 401 Midterm Exam 11/5/10 Sample Solution

Question 1. Regular expressions (20 points) In the Ada® Programming language an integer constant contains one or more digits, but it may also contain embedded underscores. Any underscores must be preceded and followed by at least one digit, which implies that a number may not begin or end with an underscore, and numbers may not contain two or more adjacent underscores. Underscores have no significance other than readability. Examples: `0`, `17`, `017`, `0_01_7`, `1_048_576`.

Integers may also be written in bases other than 10 by preceding the number with the base and a # character. For example, all of these constants represent the decimal number 15231: `15231`, `10#15231`, `16#3B7F`, `16#03B_7F`, `2#00111011_0111111`, and `8#35577`. Other bases besides powers of 2 are also allowed, for example, `2#00101`, `7#5`, and `5#10` all represent the decimal number 5.

The base of a number (preceding the # sign) must be a number that is at least 2 and at most 16, and is written in decimal. The digits in a number consist of the decimal digits 0 through 9, and the letters A, B, C, D, E, F, written only in upper case. The language specifies that the digits must be less than the base if one is given, but for this problem we'll ignore that requirement and allow any string of digits, including A through F, to appear following a base if one is present. If no base is present, the digits may only be 0 through 9.

(a) (10 points) Give a regular expression for Ada integer constants as described above. You may only use basic regular expression operators (concatenation `rs`, choice `r|s`, Kleene star `r*`) and the additional operators `r+` and `r?`. You may also specify character sets using the notation `[abw-z]`, and sets excluding specified characters `[^aeiou]`. Finally, you can name parts of the regular expression, like `vowel=[aeiou]`.

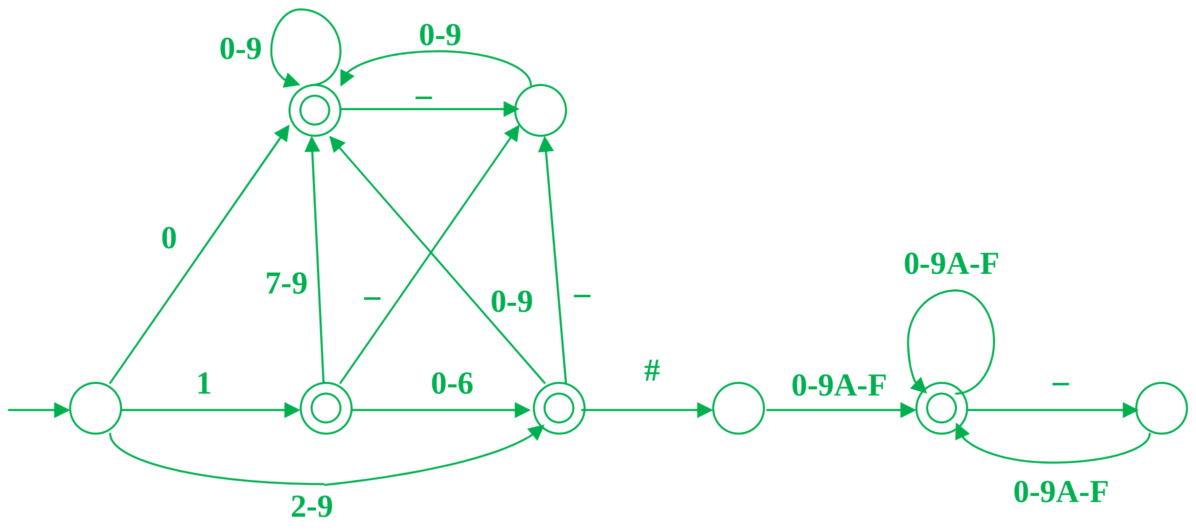
```
decint = [0-9](_?[0-9])*

basedint = (1[0-6]|[2-9])#[0-9A-F](_?[0-9A-F])*

integer = decint | basedint
```

(continued on next page)

Question 1. (cont.) (b) (10 points) Draw a DFA (Deterministic Finite Automata) that recognizes Ada integer constants as described in the problem and generated by the regular expression in your answer to part (a).

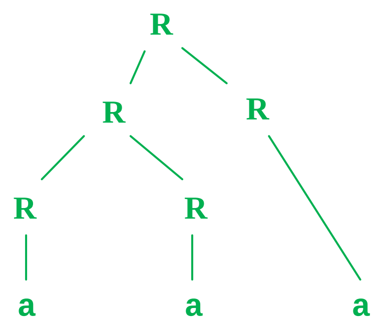


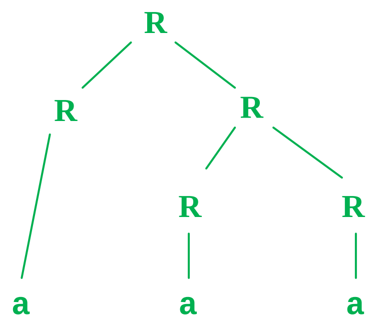
Question 2. Ambiguity (14 points) The syntax used to specify regular expressions can itself be defined by a context-free grammar. Here is one possible grammar for regular expressions with the operators concatenation, choice (|), Kleene star (*), and parenthesized subexpressions over the alphabet { a, b }.

R::= *R R*	(concatenation)
R ::= *R* | *R*	(the | here is the literal regular expression choice operator)
R ::= *R* *	(Kleene star)
R ::= (*R*)	
R ::= a	
R ::= b	

Show that this grammar for specifying the syntax of regular expressions is ambiguous.

There are a huge number of possibilities. Here are two parse trees for aaa:

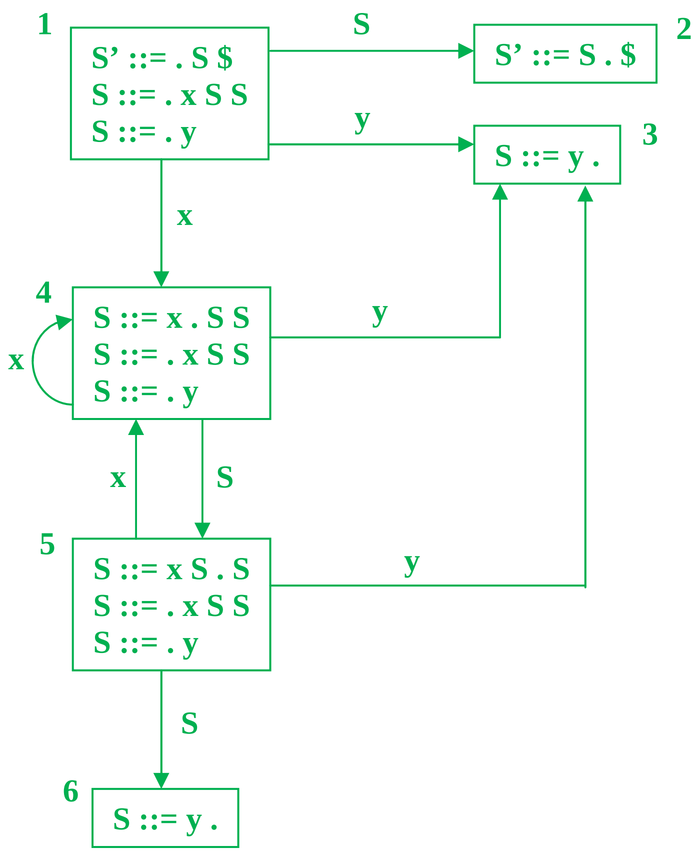




Question 3. (30 points) The you're-probably-not-surprised-to-see-it LR-parsing question. Here is a tiny grammar.

0. *S′* ::= *S* $
1. *S* ::= x *S S*
2. *S* ::= y

(a) (12 points) Draw the LR(0) state machine for this grammar.



(continued on next page)

Question 3. (cont.) Grammar repeated from previous page for reference.

0. $S' ::= S\ \$$
1. $S ::= \mathrm{x}\ S\ S$
2. $S ::= \mathrm{y}$

(b) (12 points) Construct the LR(0) parse table for this grammar based on the state machine in your answer to part (a).

	x	**y**	**$**	**S**
1	**s4**	**s3**		**g2**
2			**acc**	
3	**r2**	**r2**	**r2**	
4	**s4**	**s3**		**g5**
5	**s4**	**s3**		**g6**
6	**r1**	**r1**	**r1**	

(b) (3 points) Is this grammar LR(0)? Why or why not?

Yes. No shift-reduce or reduce-reduce conflicts in the table.

(c) (3 points) Is this grammar SLR? Why or why not?

Yes. All LR(0) grammars are also SLR.

Question 4. First/Follow/Nullable (20 points) Consider the following grammar:

S ::= *A C B*
A ::= *B C* | *B* a *A*
B ::= b *C*
C ::= *C* b | *C* c | ε

Complete the following table to give the FIRST and FOLLOW sets and NULLABLE attribute for each of the non-terminals in the grammar.

Symbol	NULLABLE	FIRST	FOLLOW
S	**F**	**{ b }**	**$**
A	**F**	**{ b }**	**{ b, c }**
B	**F**	**{ b }**	**{ a, b, c }**
C	**T**	**{ b, c }**	**{ a, b, c }**

Question 5. Parsing tools (16 points) For this problem we would like to translate arithmetic expressions from ordinary infix notation to postfix using CUP. In postfix notation a binary expression like `a*b` is written as `ab*`, with the operands appearing in the same order as they did originally followed by the operator. Operators act on the two expressions or subexpressions immediately to their left and operators are executed as soon as they are reached scanning left to right. Postfix expressions are unambiguous and do not require parentheses or operator precedence rules. Here are several examples:

Infix	Postfix	Evaluation order
`a-b+c`	`ab-c+`	a-b is computed, then c is added to the result
`a-(b+c)`	`abc+-`	a, b, and c are scanned, then b+c is computed first, then that result is the right operand of the subtraction from a
`a+b*c`	`abc*+`	multiply b*c, then add that result to a
`(a+b)*c`	`ab+c*`	add a+b first, then multiply by c
`a*b+c`	`ab*c+`	multiply a*b first, then add c
`(a*b)+c`	`ab*c+`	same
`a*b+c*d`	`ab*cd*+`	multiply a*b, multiply c*d, then add the two products

On the next page, complete the CUP specification so the resulting parser will accept infix arithmetic expressions involving identifiers; the operators +, -, and *; and parenthesized subexpressions, and print the corresponding postfix expression. (Note: you are not building a tree as you did for the parser part of the project. Just print things at appropriate places.)

You should assume that the semantic actions can call a function `print(s)` that will print the contents of string `s` to the output immediately following any previous output. The semantic action for identifier is supplied as an example. (We're taking some liberties with types here and are assuming that when an identifier token is printed the corresponding identifier string appears in the output. That is not true for the operators PLUS, TIMES, etc. You need to print appropriate literal strings in the right places.)

You will also need to add precedence declarations to handle the ambiguities in the given grammar.

Write

your

answer

on the

next

page =>

(you can detach this page if you like)

Question 5 (cont.) Complete the CUP specification below to read infix expressions and print the corresponding postfix. (Hint: the answers may be quite short and/or simple – don't be alarmed if that happens.)

```
/* Terminals (tokens returned by the scanner) */

terminal PLUS, MINUS, TIMES, LPAREN, RPAREN, IDENTIFIER;

/* Nonterminals */

nonterminal void Expr;

/* Precedence declarations – add anything appropriate here */

precedence left PLUS, MINUS;

precedence left TIMES;

/* Productions */

Expr ::= IDENTIFIER:id  {: print(id) :}

      | Expr:exp1 PLUS Expr:exp2

        {: _print("+");_ :}

      | Expr:exp1 MINUS Expr:exp2

        {: _print("-");_ :}

      | Expr:exp1 TIMES Expr:exp2

        {: _print("*"); :}

      | LPAREN Expr:exp RPAREN

        {: _/* nothing needed */_ :}

      ;
```