BITS Pilani, Dubai Campus, Second Semester 2019-2020

Course No: CS F363	III Year	Course Title: Compiler Construction
Date:8th Jun 2020	Closed Book	Weightage:15%
Duration: 70Mins.(Part A)	Comprehensive Exam	Max. Marks:30

Questions

1. Write a regular expression and draw a DFA that would determine whether a given input *binary* number is a multiple of 3 or not (*Hint:The remainder after division would always be 0,1 or 2*). **[2.5M+2.5M]**
2. Consider the instruction $e = (a/d) + (b/c) + (c/d) * (b/c)$.
   (a) Express the given instruction in three-address code form, assuming the operator '+' to be *right associative* (other precedence and associative laws holds as it is). You can use your own temporary variable names.
   (b) Rewrite the RHS of the expression (*if required*) to have a better (succinct) DAG (Directed Acyclic Graph) representation with fewer internal/ external nodes and draw the DAG.
   [2.5M+2.5M]
3. Consider the algorithm for constructing SLR parsing table, where each entry is either a $shift/reduce$ action with reference to states in the SLR automaton. What is the role of $FOLLOW()$ set in SLR parsing table construction? What exactly is the intuition behind referring to this $FOLLOW()$ set? **[5M]**
4. Consider the following basic block of three-address statements. Determine the liveness information associated with variables in each of those statements. Why do we compute this in a bottom-up fashion?(*You may use a single bit to indicate whether a variable in a statement is live or not*)

   d=b+c ;
   e=a+b ;
   b=b∗c ;
   a=e−d ;

   [5M]

5. Design a top-down predictive parser that would accept an arithmetic expression satisfying the following constraints.
   (a) An arithmetic expression is a composition of identifiers (id) and operators $\{+, *\}$, and hence the set of terminals is $\{id, +, *\}$.
   (b) The operator + is *right associative* and * is *left associative*. Further, * *has higher precedence* over +. Note that the grammar you design *should have both these constraints encoded* within itself.

   Design your grammar, eliminate left-recursion and do left-factoring if required, identify $FIRST()$ and $FOLLOW()$ sets, and design the predictive parsing table adhering to the following structure. You are supposed to fill-in the appropriate productions in the table.

Non-Terminals	Input Symbols		
	id	+	*
E1	...	...	...
...	...	...	...

   Further, add as many rows as the number of non-terminals in your grammar. **[10M]**

****

BITS Pilani, Dubai Campus, Second Semester 2019-2020

Course No: CS F363	III Year	Course Title: Compiler Construction
Date:8th Jun 2020	Closed Book	Weightage:15%
Duration:70Mins.(Part B)	Comprehensive Exam	Max. Marks:30

Questions

1. What is the role of a Lexical Analyzer Generator (LAG) in compiler design? Let us suppose you want to design a tool that would recognize binary numbers that are palindromes. Can we do this just with the help of a LAG? If yes, what would be your input to the LAG? If not, justify. **[1+3M]**
2. Consider the use of *chained symbol tables* in Block Structured Languages (BSL). Let { and } be the delimiter pair identifying block boundaries in a BSL program. During compilation, when exactly would the compiler add/delete a new block to/from the current symbol table chain? When the compiler encounters a variable in use, how does it determine whether the variable usage is valid in this scope (*declared*) or not? Justify your answer. **[4M]**
3. Answer the following questions with reference to Syntax Directed Definitions (SDD).
   (a) Distinguish between synthesized and inherited attributes.
   (b) Distinguish between S-attributed and L-attributed definitions.
   (c) Can we implement an S-attributed definition during bottom-up parsing? Justify your answer.
   (d) For the SDD given below, draw the annotated parse tree(nodes annotated with attributes and associated values) on input 8*5*3.

	Production	Semantic Rules
1)	$T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \textbf{digit}$	$F.val = \textbf{digit}.lexval$

Note: subscript 1 is just for distinguishing between instances of the same non-terminal. **[2+2+1+4M]**

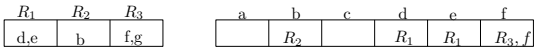
4. Consider the following segment of code:

```
int sum(int x,int y,int z)
{
    int w;
    w=x+y;
    w=w+z;
    return w;
}
main()
{
 int a=10,b=20,c=30;
 c=sum(a,b,c);
}
```

Generate the machine code equivalent to the above C segment. Explain how, various fields in an *activation record* together with *stack allocation* contributes to the above code generation (You are not expected to generate the exact machine code, neither all those fields in an activation record, but an explanation together with machine code covering all those important aspects would suffice). **[7M]**

5. Consider the given intermediate stage (from three-address statements) in an efficient (*with as few loads and stores as possible*) code generation. Illustrate the use of *register* descriptor and *address* descriptor data structures in code generation with reference to the following. For the given three-address statement, what would be the equivalent sequence of machine instructions generated? Justify your answer with reference to the notion of *write/spill penalty.* Further, update the descriptors accordingly.

. . .
. . .



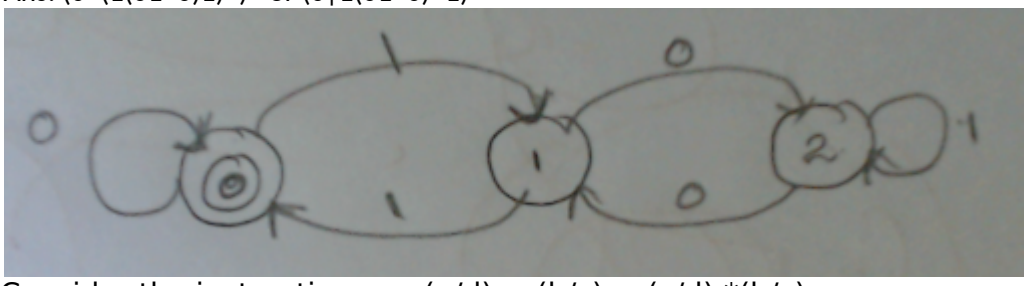
a=b+c

[6M]

****

1) Write a regular expression and draw a DFA that would determine whether a given input
binary number is a multiple of 3 or not (Hint:The remainder after division would always be
0,1 or 2). [2.5M+2.5M]

Ans: (0*(1(01*0)1)*)* or (0|1(01*0)*1)*

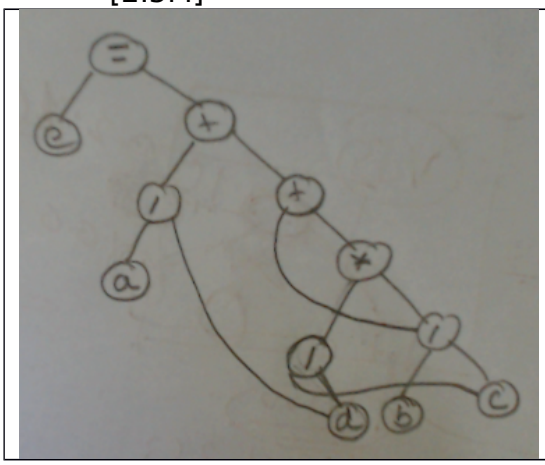


2) Consider the instruction e = (a/d) + (b/c) + (c/d) *(b/c).

(a) Express the given instruction in three-address code form, assuming the operator '+' to be right associative (other precedence and associative laws holds as it is). You can use your own temporary variable names. [2.5M]

t1 = a/d
t2 = b/c
t3 = c/d
t4 = b/c
t5 = t3 * t4
t6 = t2 + t5
t7 = t1 + t6
e = t7

(b) Rewrite the RHS of the expression to have a better (succinct) DAG (Directed Acyclic Graph) representation with fewer internal/ external nodes and draw the DAG. [2.5M]



t1 = a/d
t2 = b/c
t3 = c/d
t4 = t3 * t2
t5 = t2 + t4
t6 = t1 + t5
e = t6

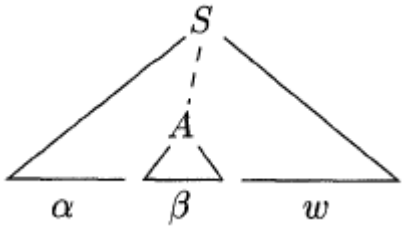
3) Consider the algorithm for constructing SLR parsing table, where each entry is either a

shift=reduce action with reference to states in the SLR automaton. What is the role of

FOLLOW() set in SLR parsing table construction? What exactly is the intuition behind

referring to this FOLLOW() set? [5M]

SLR parsing table is constructed from the LR(0) automaton. The states in the LR(0) automaton consists if items, which are grammar productions with a dot signifying the amount of input that has been seen. Eg A->X.YZ or A->XYZ. This can lead to shift-reduce conflict or reduce-reduce conflict. This can be avoided using a lookahead symbol, which are all the symbols which can appear immediately in the input. The decision to reduce is based on the current state and the next input symbol. In the SLR parsing table construction, this is equivalent to A-> β. in the automaton construction step. In the parse tree it looks as follows. The decision to reduce by the production A->β, depends on the set of symbols that immediately occur after A in any of the productions. So after β is on the stack top, and if the symbol that appears in the input is the set of symbol in the FOLLOW(A), then β is reduced to A, else the input symbol is shifted into the stack



For example E'->E, E->E+T,E->T,T->T+F,T->F,F->(E)|Id.

Follow(E) = { +,), $} and follow(T) = {+,*,),$}

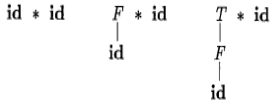
Now when the parser is in the configuration,

$T +id2$, then, T gets reduced by E->T (as + is in the follow(E))

$E +id2$

Similarly $T *id2$, then * gets shifted into the stack (as * is not in the follow(E)}

$T* id2$



Intuition of follow(A): The set of all terminals a that immediately appear after A in any parse tree derived from S. This can be computed from the grammar as follows. if there are productions where A is on the RHS, then any terminal that appears after the nonterminal A is in follow(A). If A is the rightmost nonterminal in the production, then follow of LHS nonterminal is also added to follow(A) (as seen in the figure).

Intution of follow(A): Symbols (teminals) that can come immediately after a non-terminal(A).

S -> αAw . So take first(w), if first(w) is nullable , then the what can Follow(S) will be in Follow(A).

4) Consider the following basic block of three-address statements. Determine the liveness information associated with variables in each of those statements. Why do we compute this in a

bottom-up fashion?(You may use a single bit to indicate whether a variable in a statement
is live or not)
[5M]

		dst	Op1	Op2
1	d=b+c ;	1	1	1
2	e=a+b ;	1	0	1
3	b=b*c ;	1	1	1
4	a=e-d;	1	1	1

Symbol Table

	Initial	4	3 a	3 b	2	1
a	1	0		0	1	1
b	1	1	0	1	1	1
c	1	1		1	1	1
d	1	1		1	1	0
e	1	1		1	0	0

While scanning backwards we can know when a variable was last used (assume j^{th} statement) (i.e appearing on the RHS of assignment) and then when the variable was defined (assume i^{th} statement). (ie. variable appearing on the LHS). The variable is alive from i^{th} stmt to j^{th} stmt. (with no intervening assignments to the variable). This information is also updated in the symbol table. If we scan in the forward direction, then for each variable, we have find out if it live/not live in a future statement. This search can be avoided if we go in the backward direction.

x = y + z (x in set to not live, y, z is set to live)

= x

5. Design a top-down predictive parser that would accept an arithmetic expression satisfying
the following constraints.
(a) An arithmetic expression is a composition of identifiers (id) and operators {+, *} and
hence the set of terminals is {id, + *}
(b) The operator + is right associative and * is left associative. Further, * has higher prece-
dence over +. Note that the grammar you design should have both these constraints
encoded within itself.
Design your grammar, eliminate left-recursion and do left-factoring if required, identify
FIRST() and FOLLOW() sets, and design the predictive parsing table adhering to the

following structure. You are supposed to fill in the appropriate productions in the table.

Further, add as many rows as the number of non-terminals in your grammar. [10M]

Grammar:

1) E -> T + E 2) E -> T 3) T -> T * F 4) T -> F 5) F -> id	Left Factoring: 1,2 a) E -> TE' b) E' -> +E |ϵ Left Rec: 3,4 c) T -> FT' d) T' -> *FT' | ϵ e) F -> id	First(E)=First(T)=First(F) = {id} First(E')={+, ϵ} First(T')={*, ϵ} Follow(E)=Follow(E')={$} Follow(T)=Follow(T')={+,$} Follow(F)= {*, +, $}

	id	+	*	$
E	E->TE'			
E'		E'->+E		E'-> ϵ
T	T->FT'			
T'		T'-> ϵ	T'->*FT'	T'-> ϵ
F	F->id			

BITS Pilani, Dubai Campus, Second Semester 2019-2020

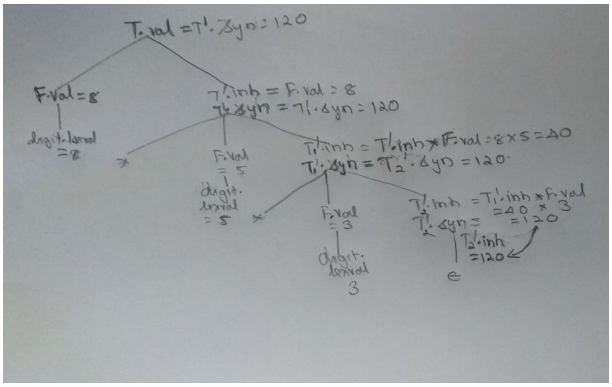
Course No: CS F363	III Year	Course Title: Compiler Construction
Date:8th Jun 2020	Closed Book	Weightage:15%
Duration:70Mins.(Part B)	Comprehensive Exam	Max. Marks:30

Questions

1. What is the role of a Lexical Analyzer Generator (LAG) in compiler design? Let us suppose you want to design a tool that would recognize binary numbers that are palindromes. Can we do this just with the help of a LAG? If yes, what would be your input to the LAG? If not, justify. **[1+3M]**
   Answer:LAG takes a formal description of the set of tokens as input and generates the required lexical analyser. The language $L_{palindrome}$ is not regular. So you can't recognize $L_{palindrome}$ with the help of a LAG.
2. Consider the use of *chained symbol tables* in Block Structured Languages (BSL). Let { and } be the delimiter pair identifying block boundaries in a BSL program. During compilation, when exactly would the compiler add/delete a new block to/from the current symbol table chain? When the compiler encounters a variable in use, how does it determine whether the variable usage is valid in this scope (*declared*) or not? Justify your answer. **[4M]**
   Answer: On {, adds a new block to the stack chain and when it sees a }, deletes the top block. For a given variable, will search for declaration in all those nodes that are there in the chain starting from top, downwards (if you write, "searches in parent node further", its incomplete).
3. Answer the following questions with reference to Syntax Directed Definitions (SDD).
   (a) Distinguish between synthesized and inherited attributes.
   Answer: Synthesized - Value is function of child nodes, and "itself".
   Inherited - Value is a function of parent, siblings and itself (no left to right ordering).
   (b) Distinguish between S-attributed and L-attributed definitions.
   Answer: S-attributed - All atttributes are synthesized.
   L-attributed - Either synthesized or inherited, where in latter, left to right ordering.
   (c) Can we implement an S-attributed definition during bottom-up parsing? Justify your answer. (Yes)
   (d) For the SDD given below, draw the annotated parse tree(nodes annotated with attributes and associated values) on input 8*5*3.

	Production	Semantic Rules
1)	$T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

Note: subscript 1 is just for distinguishing between instances of the same non-terminal. **[2+2+1+4M]**



4. Consider the following segment of code:

```
int sum(int x,int y,int z)
{
    int w;
    w=x+y;
    w=w+z;
    return w;
}
main()
{
 int a=10,b=20,c=30;
 c=sum(a,b,c);
}
```

Generate the machine code equivalent to the above C segment. Explain how, various fields in an *activation record* together with *stack allocation* contributes to the above code generation (You are not expected to generate the exact machine code, neither all those fields in an activation record, but an explanation together with machine code covering all those important aspects would suffice). **[7M]**

Answer: Points to be covered.

(a) You don't use local variable names in machine code.
(b) Each function begins with pushing the current register contents to the stack.
(c) Then allocates space for local variables.
(d) Every function call involves pushing parameters to the stack for callee. Further, push the return address and jump to the callee.
(e) The value to be returned, store in AX (eax) so that the caller can fetch it from there.
(f) Pop off memory allocated for passing parameters.
(g) Restore register values.

5. Consider the given intermediate stage (from three-address statements) in an efficient (*with as few loads and stores as possible*) code generation. Illustrate the use of *register* descriptor and *address* descriptor data structures in code generation with reference to the following. For the given three-address statement, what would be the equivalent sequence of machine instructions generated? Justify your answer with reference to the notion of *write/spill penalty*. Further, update the descriptors accordingly.

. . .
. . .

a=b+c

R_1	R_2	R_3
d,e	b	f,g

a	b	c	d	e	f
	R_2		R_1	R_1	R_3, f

[6M]

Answer: Spill penalty of R_1 is 2 (assuming d and e are live at this point) and that of R_3 is at most 1, depending on whether g is live or not. Further, whether you don't store a back to memory. It remains in the register itself. Further, the target register for a, is either R_2 or R_3 (depending on liveness). You have to mention your assumptions. Your machine code would contain one load instruction and one add instruction.

****