Birla Institute of Technology and Science, Pilani

Department of Computer Science and Information Systems

Compiler Construction (CS F363)

Second Semester 2021-22

Mid Semester Test

Part 2 (Closed Book)

Date: March 12, 2022
Day: Saturday

Duration:45 minutes
Max. Marks: 30

Instructions: Write neatly using a blue or black pen on the answer sheet given. Answers written with pencil will not be considered for evaluation. Over written answers will not be considered for reevaluation.

Q1. A language supports four different representations of integer numbers-binary, decimal, octal and hexadecimal. Examples of these are B011001111, D342001696, O27104 and H001A90F1 respectively. The base of these systems are 2, 10, 8 and 16 respectively and the numbers use prefixes B, D, O and H respectively. Design a deterministic finite automaton (DFA) that accepts lexemes and tokenizes them as TK_BIN, TK_DEC, TK_OCT and TK_HEX respectively. Ensure that the DFA has single start state. **[4 M]**

Q2. Consider the following grammar G = (N, T, P, S) where S=F is the start symbol and terminals T={id, (,), *, COMMA}

F → T id (Parameters)
T → id
T → T*
Parameters → ParList
Parameters → ε
ParList → T id COMMA ParList
ParList → T id

a) Is the given Grammar LL(1)? Justify. If no, then transform this CFG to LL(1) compatible CFG. **[3 M]**
b) Compute the First and Follow sets of all the non-terminals and terminals in the modified grammar obtained in previous step a). **[2 M]**
c) Construct the LL(1) Parsing table clearly mentioning all the entries related to production rules and erroneous entries. **[4 M]**
d) Show step-by-step process of parsing of input string id** id(id*** id COMMA id*id) as per the following pattern discussed in the class. **[3M]**

Stack	Input	Action
$F	id** id(id*** id COMMA id*id)	-
	-	-

Q3. Consider the already augmented Grammar with S as the new start symbol and terminals are {;, =, id, [,], (,)}

(0) S → Stmts $
(1) Stmts → Stmt
(2) Stmts → Stmts ; Stmt
(3) Stmt → V = Exp
(4) V → id [Exp]
(5) V → id
(6) Exp → id
(7) Exp → (Exp)

a) Construct the canonical collection of LR(1) items and the DFA capable of recognizing it. **[6M]**
b) Design the CLR(1) parsing table clearly highlighting all the action and Goto entries and determine if this grammar is CLR(1) or not. Justify. **[4M]**
c) In addition, create the LALR(1) parsing table and find out the conflicts (if exist, any) **[4M]**

****

Birla Institute of Technology and Science, Pilani

Department of Computer Science and Information Systems

Compiler Construction (CS F363)

Second Semester 2021-22

Comprehensive Examination

Part 2 (Closed Book)

Date: May 14, 2022 **Duration:60 minutes**

Day: Saturday **Max. Marks: 30**

Instructions: Write neatly using a blue or black pen on the answer sheet given. Answers written with pencil will not be considered for evaluation. Over written answers will not be considered for reevaluation. Return this question paper along with the answer sheet.

Q1. Consider the following high level C-like loop construct which uses short circuit evaluation of the Boolean expressions. **[2 + 2+ 2 + 3 + 3 + 4= 16 M]**

```
a = (b + c) – d;
while (a<=b && p<q || a>d) {
    c = a - b*c + (a-c);
    if (x >= y)
        c = p*c - b*m + a;
    else
        a = b - c*a+ b*a;
    p = m + a – b;
}
```

(a) Construct the equivalent three address code for the above input using appropriate semantic rules. [Choose Labels and temporaries from the pools L0, L1, L2, ... and t1, t2, t3, ... respectively]

(b) Identify leader statements and construct the basic blocks for the three address code generated above.

(c) Construct the Control Flow Graph (CFG) for the above three address code.

(d) Compute the liveness of variables at each node of the CFG using the data flow equations discussed in the class. Ensure convergence of the process. [Show the iterations of the computations in appropriately drawn tables]

(e) Use registers R1, R2, R3, R4 and R5, of which R1 and R2 should be used across the basic block in the loop while remaining three registers should be used for the basic block. Use *global register allocation* method to identify the variables which are assigned the registers R1 and R2.

(f) Use ***getreg()*** algorithm to select the registers and generate the assembly code so as to minimize the redundant load and store operations within the basic blocks.

Q2. Consider the following CFG [{S, A, X, B, D, I}, {c, l, x, v, i}, P, S] which generates all the lower-case roman numbers in the range of 1-99.

1. $S \rightarrow xAB$
2. $S \rightarrow X$
3. $D \rightarrow v$
4. $A \rightarrow c$
5. $I \rightarrow \epsilon$
6. $X_1 \rightarrow xX_2$
7. $X \rightarrow B$
8. $S \rightarrow lX$
9. $B \rightarrow iD$
10. $A \rightarrow l$
11. $B \rightarrow vI$
12. $D \rightarrow x$
13. $I_1 \rightarrow iI_2$
14. $B \rightarrow I$

Write the semantic rules associated with the above grammar rules to calculate the final decimal value of roman numeral in S.value. You are allowed to use only one synthesized attribute (i.e. value) for the creation of the semantic rules.
Note, while writing your answers you should follow the following format:

Production Rule No. { Attribute Equation(s)}

1. **{Attribute Equation(s)}**
2. **{Attribute Equation(s)}**

and so on.....

[**HINT:** Roman numerals constitutes symbols like i (1), v (5), x (10), l (50), c (100). A smaller number placed adjacent to a large number implies one must subtract the least number from the largest number (e.g. *xc means 90*). Only one number can be subtracted (such as, iiv does not imply 3). On the other hand, a number placed after a number with equal or greater value is added to it (such as, cc implies 200.)

[14 M]

****

Birla Institute of Technology and Science, Pilani

Department of Computer Science and Information Systems

Compiler Construction (CS F363)

Second Semester 2021-22

Compiler project: Online examination

(Open Book)

Date: April 24, 2022
Day: Sunday

Duration:180 minutes
Max. Marks: 45

Q1. Introduce a new keyword printoffset to be used with a variable identifier as printoffset(c2b3) and modify your compiler to handle this special output statement. This accesses the symbol table and prints the value of the computed offset of the variable c2b3 stored in the symbol table. Your code generator should produce an assembly equivalent of the above statement to print on the console the offset of the variable used when code.asm is executed. You can use the test case q1.txt to verify your code. *[5 Marks]*

Q2. Implement the following semantic rules to the existing language. You can use the test cases q2_parts_a_and_b.txt and q2_parts_c_and_d.txt to verify your code for the given parts below. *[2+2+2+2 = 8 Marks]*

(a) The language supports structural and name type equivalence both for the record type variables.

(b) The language relaxes the constraint on use of recursive call and allows the use of maximum three recursive calls in a function, but detects the fourth or later recursive calls as errors.

(c) Count the total number of conditional variables used in the while loop which are not assigned any values within the body of the loop. Add option number 12 in your driver to print line number wise details of the count. Use the line numbers of the first and the last statements of the corresponding while loop and print the count in front of the line numbers appropriately. Note that you just have to count the numbers here but while semantics still remains the same.

(d) Detect the presence of more than one union type fields in the variant record and report this as an error

Q3. Introduce a new construct of C-like switch-case statement in the given language. The use of break statement is optional and the type of switch variable can be only an integer. Report error if the type of the switch variable (e.g c2) is real or record type. Example code in the given language is

```
switch (c2)
        case 0: <statements>; break;
        case 2: <statements>; break;
        case 3: <statements>; break;
        default: <statements>;
endswitch
```

The syntax of the above necessarily requires minimum of two case statements and a default statement at the end.

Implement the following as asked. *You can use the test case q3.txt to verify your code.* ***[2+3+2+2+2+3= 14 Marks]***

(a) Modify your lexical analyzer to incorporate the new patterns used in the above construct.

(b) Modify your syntax analyzer to implement the above construct.

(c) Modify your Abstract Syntax Tree construction for the above construct.

(d) The type of the switch variable is integer. Report error otherwise.

(e) Compute the total number of case statements in the given switch case statement (including the default statement). Add option number 11 in your driver to show this count.

(f) Modify your code generator to produce code in assembly language for the above construct.

Q4. The language supports string data type in addition to the data types already existing in the given language. The string type is defined using the keyword *string* followed by the maximum allowed number of characters in it. The variable identifiers of string data type can store only the static string values such as $compiler or $construction. The string variables cannot be read at run time, therefore the user is expected to provide the string values in the source code. This means that read(c2c6) is not valid, if the variable type is string. To illustrate, the variables c2c6 and d4d555 declared to be of string type can be initialized as follows.

```
type string[25 ] :c2c6;
type string[15] : d4d555;
```

c2c6 <--- $compiler_;
d4d555<--- $construction;

Lexical units: The lexical pattern for string constant is represented by the regular expression [$][a-z][a-z _]* which cannot accept any uppercase letter or any other symbol other than the lower case alphabet starting with a $ sign. The pattern can also include underscore in between any number of times after the first lowercase alphabet. This means the string $a_b_c is valid, but $_abc is not. The accepted lexeme representing the string value is tokenized as TK_STR while the type string is tokenized as TK_STRING. A new operator with symbol @, which is tokenized as TK_LEN is used to get the length of the string as is described below. The string value tokenized as TK_STR cannot participate in any computation directly and cannot be part of any expression. Rather it can be only used to initialize a variable of string type as is shown above in the example.

Data type: The type of the string value (e.g. $compiler_) is *string*. As all other types, a string variable can be used as the type of a field in a record or a union definition. The function parameters can be of type strings like other types supported in the language. The string variables can undergo only two types of operations – concatenation and string length. The concatenation is implemented using the + (plus operator) and the string length is obtained by the unary operator @ preceding the name of the variable.

Concatenation operator: Consider a statement with the string variables on the right hand side of the expression.

c2c6<--- c2c6 + d4d555 ;

In this, the resultant concatenated string $compiler_construction will be stored in the location bound to the left hand side variable c2c6. Since the types of the variables d4d555 and c2c6 are same and are of string type, which also matches with the type of c2c6, the expression and the statement both are type correct. Note that the sizes associated with the types of the string variables do not cause any type mismatch as long as the variables are of string type. Instead, the size is used only for the width and offsets computation. However, the variable on the left hand side is required to be of the size to accommodate the new string obtained as a result of concatenation.

Length operator: The length operator @ is applied only to a string type variable as follows.

d5 <--- @c2c6;

The type of the left hand side variable identifier can be only integer. If the type of d5 is not an integer, then this will be reported as an error. In addition, the error is reported if the type of the variable argument of the operator @ is not of string type. Another point to note is that @ cannot be applied to the string value, for example @$compiler is not valid, but @c2c6 is valid, where c2c6 is a variable identifier. A variable of string type cannot be added with any variable of any other data type. The operators of minus, multiplication or division cannot have an operand of type string. It is also essential for a user to initialize a variable of string type before its use in any expression or a statement.

Storage details for string constants and data of string type: The string lexeme $of comprises of characters o and f, while $ is not part of the string characters when the string constant is stored in memory. A character in string variable uses only one byte space and the space for each string is aligned to the 4 bytes memory. To understand this, $compiler is a string comprising of 8 characters requiring 8 bytes of space. The $ is not stored anywhere in the memory. Similarly, the string $programming consists of 11 characters requiring 11 bytes of space is aligned in a total of 12 bytes. For example, the string $abcde uses five characters and requires 8 bytes after aligning the 5 bytes to the nearest multiple of four bytes i.e. 8 bytes. Two strings cannot share the same block of four bytes which means that if there is another string $pqrst requiring 5 bytes, it cannot use the remaining empty space of the previous string. Instead, it will use fresh 8 bytes for storing the characters p, q, r, s and t. A string variable (say c2c4) of string type (say string[10]) can store a string value (say $compiler) only if the length of the variable c2c3 (i.e. @c2c3) is less than or equal to the string size 10. To recall, the length is calculated excluding the dollar sign.

Implement the following**. You can use the test case q4.txt to verify your code.* ***[3+4+8+3 = 18 Marks]

(a) Modify your lexical analyzer to recognize the new patterns of string constant, string type, and the length operator.
(b) Modify the grammar rules and incorporate changes to implement the new constructs as described above.
(c) Modify the type checker and semantic analyzer to incorporate the above specifications. The symbol table captures the new construct and stores the scope, types, widths and offsets of the variables of all types including the above.
(d) Modify the code generator to incorporate the above changes and produce results.

****

group 11

Birla Institute of Technology & Science, Pilani

Pilani Campus

I Semester / II Semester / Summer Term 20___ - 20___

Mid-Semester (Regular/Make-Up)

ID No. 2019A7PS0067P Name RUCHIR JAIN

Course No. CS F363 Course Title COMPILER CONSTR. Section No. 1

Instructor's Name VANDANA A. Room No. 1226 Date 12.8.22.

No. of Supplementary Copies Attached 0

Verified: Signature of Invigilator:

Question No.	Marks Obtained	Student's request for rechecking with remarks	Examiner's remarks
1	4		
2	12		
3	0.5		
4			
5			
6			
7			
8			
9			
10			
Total	(in figures) 16.5	(in words)	Examiner's Signature

INSTRUCTIONS

1. Enter all the required details on the cover of every answer booklet.
2. Write on both sides of the sheet in the answer book. Rough work, if any should be done at the bottom of the page. Finally cross out the rough work and draw a horizontal line to separate it from the rest of the material on the page. Also, cross out all blank pages in the answer booklet.
3. Any answer crossed out by the student will not be examined by the examiner.
4. No sheet should be torn from the answer booklet.
5. Mobile phones or any electronic communication/storage device of any kind is prohibited in the examination hall.
6. Use of any unfair means will make the candidate liable to disciplinary action.
7. Student should not leave the examination hall without submitting the answer booklet to invigilator on duty.
8. Student must abide by all the instructions given by the invigilator(s) on duty.

I have carefully read and understood all the instructions.

I do understand that any attempt to use unfair means of any kind in an examination is a serious and punishable offence.

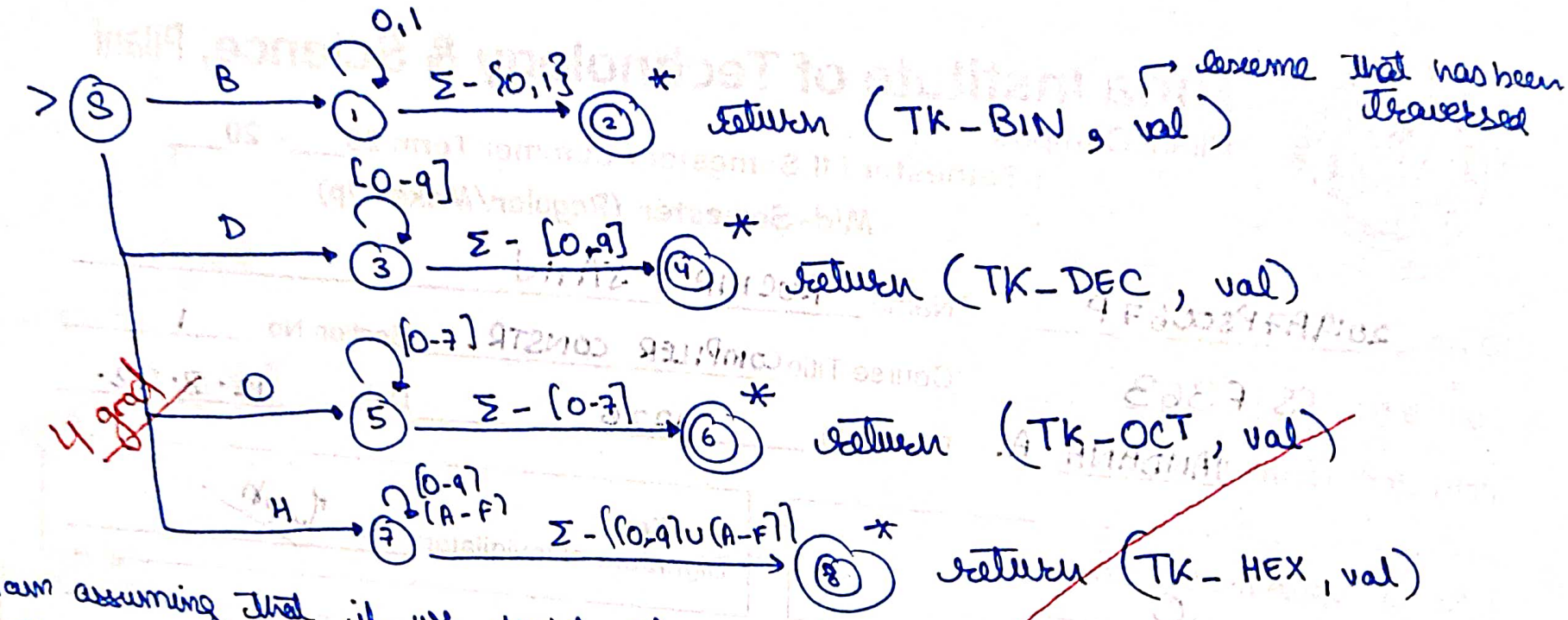
I hereby declare that I will not attempt to do any malpractice in the examination.

Ruchir

Signature of the student

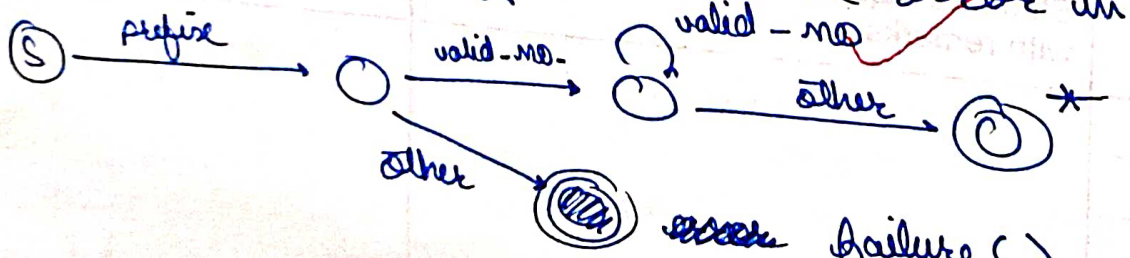
Part 2

Q1.



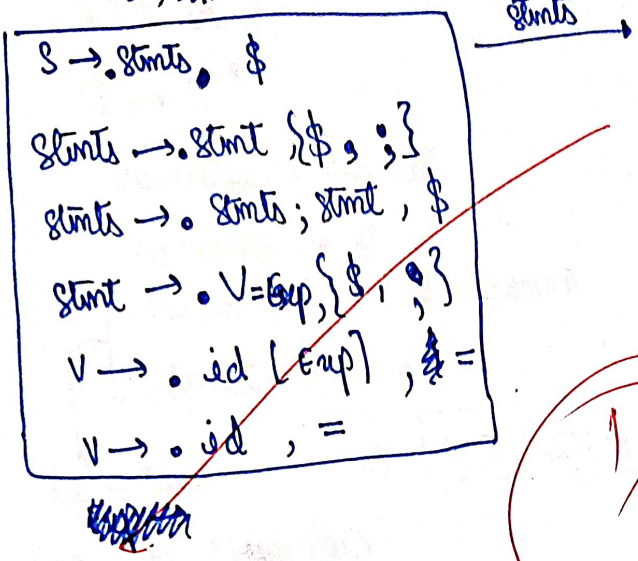
4 grad

Here I am assuming that if we don't get a valid number after prefix but get another character, then it will be flagged as lexical error in implementation. Otherwise do



for each prefix B, D, O, H.

Q3.


stmts
S → . stmts , $
stmts → . stmt {$, ;}
stmts → . stmts ; stmt , $
stmt → . V = exp , {$, ;}
V → . id [exp] , =
V → . id , =

1/2

PTO →

Q2.(a) F → T id (Parameters)

T → id

T → T*

Parameters → ParList

Parameters → ∈

ParList → T id COMMA ParList

ParList → T id

⇒ Not LL(1)

Left factored

F → T id (Parameters)

T → id | T*

Parameters → ParList | ∈

ParList → T id A

A → COMMA ParList | ∈

For T → T* left recursion β = id α = *

⇒ T → id T'

T' → * T' | ∈

3

⇒

F → T id (Parameters)

T → id T'

T' → * T' | ∈

Parameters → ParList | ∈

ParList → T id A

A → COMMA ParList | ∈

→ first(* T') = * ; follow(T') = follow(T) = id ⇒ Disjoint

→ first(ParList) = first(T) = id ; follow(Parameters) =) ⇒ Disjoint

→ First(COMMA ParList) = COMMA ⇒ Disjoint

follow(A) = follow(ParList) = follow(Parameters) =)

⇒ Above is LL(1) compatible

NT	first	follow
F	id	$
T	id	id
T'	*, ∈	id
Parameters	id, ∈	)
ParList	id	)
A	COMMA, ∈	)

PTO →

(c)

	id	(	)	*	COMMA	$
F	F→T id (Params)					
T	T→ id T'					
T'	T'→ ∈			T'→ *T'		
Parameters	Parameters→ Parlist		Parameters→ ∈			
Parlist	Parlist→T id A					
A			A→ ∈		A→ COMMA Parlist	

F→ T id (Params)
T→ id T'
T'→ *T'
T'→ ∈
Parameters → Parlist
Parlist → T id A
A→ COMMA Parlist
A→ ∈

$) A id T' *	* id) $	Pop & ip++
$) A id T'	id) $	T'→ ∈
$) A id	id) $	Pop & ip++
$) A	) $	A→ ∈
$)	) $	Pop & ip++
$	$	Accept

(d)

Stack	Input	Action
$ F	id * * id (id * * id COMMA id * id) $	F→T id (Params)
$) Params (id T	"	T→ id T'
$) Params (id T' id	"	Pop & ip++
$) Params (id T'	* * id ( )	T'→ *T'
$) Params (id T' *	* * id ( )	Pop & ip++
$) Params (id T'	* id ( )	T'→ *T'
$) Params (id T' *	* id ( )	Pop & ip++
$) Params (id T'	id ( )	T'→ ∈
$) Params (id	id ( )	Pop & ip++
$) Params (	( )	Pop & ip++
$) Params	id * * id COMMA id * id)	Params→ Parlist
$) Parlist	"	Parlist→ T id A
$) A id T	id * * id COMMA id * id)	T→ id T'
$) A id T' id	id	Pop & ip++
$) A id T'	* * id ...	T'→ *T'
$) A id T' *	* * id ...	Pop & ip++
$) A id T'	* id ...	T'→ *T'
$) A id T' *	* id ...	Pop & ip++
$) A id T'	id COMMA id * id) $	Pop & ip++
$) A id	COMMA id * id) $	A→ COMMA Parlist
$) A	COMMA "	Pop & ip++
$) Parlist COMMA	id * id) $	Parlist→ T id A
$) Parlist	id * id) $	T→ id T'
$) A id T	id * id ...	Pop & ip++
$) A id T' id	* id) $	T'→ *T'
$) A id T'		

Team group 11

Tushar Garg - 2019A7PS0104P

Usneek Singh - 2019A7PS0127P

Vikas Balani - 2019A7PS0054P

Ruchir Jain - 2019A7PS0067P

Abhijith S Raj - 2019A7PS0055P

Modified Grammar Rules

<typeDefinition> ===> TK_RECORD TK_RUD <fieldDefinitions> TK_ENDRECORD | <definetypestmt>

<fieldDefinition> ===> TK_TYPE <dataType> TK_COLON. TK_FIELDID TK_SEM

<singleOrRecId> ===> TK_ID <c>

<c> ===> TK_DOT TK_FIELDID <c> | ∈

<conditionalStmt> ===> TK_IF <booleanExpression> TK_THEN <stmt> <otherStmts> <M>

<M> ===> TK_ENDIF | TK_ELSE <otherStmts> TK_ENDIF

<arithmeticExpression> ===> <first> <S>

<S> ===> TK_PLUS <arithmeticExpression> | TK_MINUS <arithmeticExpression> | ∈

<first> ===> <second> <T>

<T> ===> TK_MUL <first> | TK_DIV <first> | ∈

<second> ===> TK_OP <arithmeticExpression> TK_CL | <var>

TK_ENDWHILE, TK_ENDIF, TK_ELSE

S. No.	Non Terminal	First Set	Follow Set
15.	<fieldDefinitions>	TK-TYPE	TK-ENDRECORD , TK-ENDUNION
1.	<program>	TK-FUNID , TK-MAIN	$
2.	<mainFunction>	TK-MAIN	$
3.	<otherFunctions>	TK-FUNID , ∈	TK-MAIN
4.	< function >	TK-FUNID	TK-FUNID, TK-MAIN
5.	<input-par>	TK-INPUT	TK-OUTPUT , TK-SEM
6.	<output-par>	TK-OUTPUT, ∈	TK-SEM
7.	<parameter_list>	TK-INT, TK-REAL, TK-RECORD, TK-UNION	TK-SQR
8.	<dataType>	TK-INT, TK-REAL, TK-RECORD, TK-UNION	TK-ID, TK-COLON
9.	<primitiveDatatype>	TK-INT , TK-REAL	TK-ID , TK-COLON
10.	<constructedDatatype>	TK-RECORD, TK-UNION	TK-ID , TK-COLON
11.	<remaining_list>	TK-COMMA , ∈	TK-SQR
12.	<stmts>	TK-RECORD, TK-UNION, TK-TYPE , TK-DEFINETYPE, TK-ID , TK-WHILE , TK-IF, TK-READ , TK-WRITE, TK-CALL , TK-SQL, TK-RETURN	TK-END
13.	<typeDefinitions>	TK-RECORD , TK-UNION, TK-DEFINETYPE, ∈	TK-TYPE , TK-ID, TK-WHILE , TK-IF, TK-READ , TK-WRITE , TK-CALL, TK-SQL , TK-RETURN , TK-ENDWHILE, TK-ENDIF , TK-ELSE
14.	<typeDefinition>	TK-RECORD , TK-UNION, TK-DEFINETYPE	TK-RECORD, TK-UNION, TK-TYPE, TK-DEFINETYPE , TK-ID , TK-WHILE, TK-IF , TK-READ , TK-WRITE, TK-CALL , TK-SQL , TK-RETURN, TK-ENDWHILE, TK-ENDIF , TK-ELSE

15.	<fieldDefinitions>	TK-TYPE	TK-ENDRECORD , TK-ENDUNION
16.	<fieldDefinition>	TK-TYPE	TK-TYPE, TK-ENDRECORD, TK-ENDUNION
17.	<moreFields>	TK-TYPE, ∈	TK-ENDRECORD', TK-ENDUNION
18.	<declarations>	TK-TYPE, ∈	TK-ID, TK-WHILE, TK-IF, TK-REA TK-WRITE, TK-CALL, TK-SQL, TK-RETURN, TK-ENDWHILE, TK-ENDIF, TK-ELSE
19.	<declaration>	TK-TYPE	TK-TYPE, TK-ID, TK-WHILE, TK-IF, TK-READ, TK-WRITE, TK-CALL, TK-SQL, TK-RETURN TK-ENDWHILE, TK-ENDIF, TK-ELSE
20.	<global_or_not>	TK-GLOBAL, ∈	TK-SEM
21.	<otherStmts>	TK-ID, TK-WHILE, TK-IF, TK-RETURN, TK-ENDWHILE, TK-READ, TK-WRITE, TK-ENDIF, TK-ELSE TK-CALL, TK-SQL, ∈	
22.	<stmt>	TK-ID, TK-WHILE, TK-IF, TK-READ, TK-WRITE, TK-CALL, TK-SQL	TK-ID, TK-WHILE, TK-READ, TK-WRITE, TK-CALL, TK-SQL, TK-RETURN, TK-ENDWHILE, TK-ENDIF, TK-ELSE
23.	<assignmentStmt>	TK-ID	TK-ID, TK-WHILE, TK-IF, TK-READ, TK-WRITE, TK-CALL, TK-SQL, TK-RETURN, TK-ENDWHIL TK-ENDIF, TK-ELSE
24.	<singleOrRecId>	TK-ID	TK-ASSIGNOP

25.	<funCallStmt>	TK-CALL, TK-SQL	TK-ID, TK-WHILE, TK-IF, TK-READ, TK-WRITE, TK-CALL, TK-SQL, TK-RETURN, TK-ENDWHILE, TK-ENDIF, TK-ELSE
26.	<outputParameters>	TK-SQL, ∈	TK-CALL
27.	<inputParameters>	TK-SQL	TK-ID, TK-WHILE, TK-IF, TK-READ, TK-WRITE, TK-CALL, TK-SQL, TK-RETURN, TK-ENDWHILE, TK-ENDIF, TK-ELSE
28.	<iterativeStmt>	TK-WHILE	TK-ID, TK-WHILE, TK-IF, TK-READ, TK-WRITE, TK-CALL, TK-SQL, TK-RETURN, TK-ENDWHILE, TK-ENDIF, TK-ELSE
29.	<conditionalStmt>	TK-IF	TK-ID, TK-WHILE, TK-IF, TK-READ, TK-WRITE, TK-ENDWHILE, TK-ENDIF, TK-ELSE
30.	<ioStmt>	TK-READ, TK-WRITE	TK-ID, TK-WHILE, TK-IF, TK-READ, TK-WRITE, TK-ENDWHILE, TK-ENDIF, TK-ELSE
31.	<arithmeticExpression>	TK-OP, TK-ID, TK-NUM, TK-RNUM	TK-CL, TK-SEM
32.	<booleanExpression>	TK-OP, TK-NOT, TK-ID, TK-NUM, TK-RNUM	TK-CL, TK-THEN

33.	<var>	TK-ID, TK-NUM, TK-RNUM	TK-CL, TK-MUL, TK-DIV, TK-PLUS, TK-MINUS, TK-SEM, TK-LT, TK-LE, TK-EQ, TK-GT, TK-GE, TK-NE, TK-THEN
34.	<logicalOp>	TK-AND, TK-OR	TK-OP
35.	<relationalOp>	TK-LT, TK-LE, TK-EQ, TK-GT, TK-GE, TK-NE	TK-ID, TK-NUM, TK-RNUM
36.	<returnStmt>	TK-RETURN	TK-END
37.	<optionalReturn>	TK-SQL, ∈	TK-SEM
38.	<idList>	TK-ID	TK-SQR
39.	<more-ids>	TK-COMMA, ∈	TK-SQR
40.	<definetypestmt>	TK-DEFINETYPE	TK-RECORD, TK-UNION, TK-TYPE, TK-DEFINETYPE, TK-ID, TK-WHILE, TK-IF, TK-READ, TK-WRITE, TK-CALL, TK-SQL, TK-RETURN, TK-ENDWHILE, TK-ENDIF, TK-ELSE
41.	<A>	TK-RECORD, TK-UNION	TK-RUID
42.	<C>	TK-DOT, ∈	TK-ASSIGNOP
43.	<M>	TK-ENDIF, TK-ELSE	TK-ID, TK-WHILE, TK-IF, TK-READ, TK-WRITE, TK-CALL, TK-SQL, TK-RETURN, TK-ENDWHILE, TK-ENDIF, TK-ELSE

44.	<S>	TK-PLUS, TK-MINUS, ∈	TK-CL, TK-SEM
45.	<first>	TK-OP, TK-ID, TK-NUM, TK-RNUM	TK-PLUS, TK-MINUS, TK-CL, TK-SEM
46.	<second>	TK-OP, TK-ID, TK-NUM, TK-RNUM	TK-MUL, TK-DIV, TK-PLUS, TK-MINUS, TK-CL, TK-SEM
47.	<T>	TK-MUL, TK-DIV, ∈	TK-PLUS, TK-MINUS, TK-CL, TK-SEM

Note: The <operator> non-terminal has been eliminated and the production rules associated with <arithmetic Expression> have been modified to enforce precedence amongst arithmetic operators.