Semantic Rules

Group 18:

Madhav Gupta (2019A7PS0063P)
Meenal Gupta (2019A7PS0243P)
Pratham Gupta (2019A7PS0051P)
Sankha Das (2019A7PS0029P)
Yash Gupta (2019A7PS1138P)

Functions used in the implementation of the semantic rules:

```
child(i, <node>) {
      /*
            returns a pointer to the i^th child of the parse tree node pointed
            to by <node>
      /*
}

child(i … j, <node>) {
      /*
            returns pointers for the i^th to j^th children of the parse tree node
            pointed to by <node>
      /*
}

makeNode(nodeLabel, <child>, <child>, …) {
      /*
            makes a node for the AST. The node is a structure which has
            attributes name (= nodeLabel) and the children of the node
            (= the <child> pointers). The function returns a pointer to the
            created AST node
      */
}

makeLeaf(nodeLabel, <value>) {
      /*
            makes a leaf node for the AST. The node is a structure which
            has attributes name (= nodeLabel) and the value of the entity
            represented by the node (Eg. numerical values, etc). The
            function returns a pointer to the created AST Leaf node.
      */
}
```

List of Semantic Rules:

1. **<program> ===> <otherFunctions> <mainFunction>**

```
{
        <program>.ptr = makeNode(
                                “Program”,
                                <otherFunctions>.ptr,
                                <mainFunction>.ptr
                        );
        free(child(1…2, <program>));
}
```

2. **<mainFunction> ===> TK_MAIN <stmts> TK_END**

```
{
        <mainFunction>.ptr = makeNode(“MainFunc”, <stmts>.ptr);
        free(child(1...2, <mainFunction>));
}
```

3. **<otherFunctions> ===> <function> <otherFunctions>**

```
{
        <otherFunctions>.ptr = makeLinkedListNode(“FuncLinkedListNode”);
        <otherFunctions>.ptr.data = <function>.ptr;
        <otherFunctions>.ptr.next = <otherFunctions>.ptr;
        free(child(1...2, <otherFunctions>));
}
```

4. **<otherFunctions> ===> epsilon**

```
{
        <otherFunctions>.ptr = NULL;
}
```

5. **<function> ===> TK_FUNID <input_par> <output_par> TK_SEM <stmts> TK_END**

```
{
        <function>.ptr = makeNode("FuncDef",
                                  makeLeaf("FuncId", child(1, <function>).entry),
                                  <input_par>.ptr,
                                  <output_par>.ptr,
                                  <stmts>.ptr
                         );
        free(child(1...6, <function>));
}
```

6. **<input_par> ===> TK_INPUT TK_PARAMETER TK_LIST TK_SQL <parameter_list> TK_SQR**

```
{
        <input_par>.ptr = <parameter_list>.ptr;
        free(child(1...6, <input_par>));
}
```

7. **<output_par> ===>TK_OUTPUT TK_PARAMETER TK_LIST TK_SQL <parameter_list> TK_SQR**

```
{
        <output_par>.ptr = <parameter_list>.ptr;
        free(child(1…6, <output_par>));
}
```

8. **<output_par> ===> epsilon**

```
{
        <output_par>.ptr = NULL;
}
```

9. **<parameter_list> ===> <dataType> TK_ID <remaining_list>**

```
{
        <parameter_list>.ptr = makeLinkedListNode("ParameterLinkedListNode");
        <parameter_list>.ptr.data = makeNode(
                                                "Parameter",
                                                <dataType>.ptr,
                                                child(2, <parameter_list>).entry
                                        );
        <parameter_list>.ptr.next = <remaining_list>.ptr;
        free(child(1...3, <parameter_list>));
}
```

10. **<dataType> ===> <primitiveDatatype>**

```
{
        <dataType>.ptr = <primitiveDatatype>.ptr;
        free(child(1, <dataType>));
}
```

11. **<dataType> ===> <constructedDatatype>**

```
{
        <dataType>.ptr = <constructedDatatype>.ptr;
        free(child(1, <dataType>));
}
```

12. **<primitiveDatatype> ===> TK_INT**

```
{
        <primitiveDatatype>.ptr = makeLeaf("typeINT");
        free(child(1, <primitiveDatatype>));
}
```

13. **<primitiveDatatype> ===> TK_REAL**

```
{
        <primitiveDatatype>.ptr = makeLeaf("typeREAL");
        free(child(1, <primitiveDatatype>));
}
```

14. <constructedDatatype> ===> TK_RECORD TK_RUID

```
{
        <constructedDatatype>.ptr = makeLeaf(
                                        "typeRECORD",
                                        child(2,<constructedDatatype>).entry
                                    );
        free(child(1…2, <constructedDatatype>));
}
```

15. <constructedDatatype> ===> TK_UNION TK_RUID

```
{
        <constructedDatatype>.ptr = makeLeaf(
                                        "typeUNION",
                                        child(2, <constructedDatatype>).entry
                                    );
        free(child(1…2, <constructedDatatype>));
}
```

16. <constructedDatatype> ===> TK_RUID

```
{
        <constructedDatatype>.ptr = makeLeaf(
                                        "RecUnionId",
                                        child(1, <constructedDatatype>).entry
                                    );
}
```

17. <remaining_list> ===> TK_COMMA <parameter_list>

```
{
        <remaining_list>.ptr = <paramater_list>.ptr;
        free(child(1...2, <remaining_list>));
}
```

18. <remaining_list> ===> epsilon

```
{
        <remaining_list>.ptr = NULL;
}
```

19. <stmts> ===> <typeDefinitions> <declarations> <otherStmts> <returnStmt>

```
{
        <stmts>.ptr = makeNode(
                                “Stmts”,
                                <typeDefinitions>.ptr,
                                <declarations>.ptr,
                                <otherStmts>.ptr,
                                <returnStmt>.ptr
                        );
        free(child(1...4, <stmts>));
}
```

20. <typeDefinitions> ===> <actualOrRedefined> <typeDefinitions>$_1$

```
{
        <typeDefinitions>.ptr = makeLinkedListNode(“TypeDefLinkedListNode”);
        <typeDefinitions>.ptr.data = <actualOrRedefined>.ptr;
        <typeDefinitions>.ptr.next = <typeDefinitions>1.ptr;
        free(child(1...2, <typeDefinitions>));
}
```

21. <typeDefinitions> ===> epsilon

```
{
        <typeDefinitions>.ptr = NULL;
}
```

22. <actualOrRedefined> ===> <typeDefinition>

```
{
        <actualOrRedefined>.ptr = <typeDefinition>.ptr;
        free(child(1, <actualOrRedefined>));
}
```

23. <actualOrRedefined> ===> <definetypestmt>

```
{
        <actualOrRedefined>.ptr = <definetypestmt>.ptr;
        free(child(1, <actualOrRedefined>));
}
```

24. <typeDefinition> ===> TK_RECORD TK_RUID <fieldDefinitions> TK_ENDRECORD

```
{
        <typeDefinition>.ptr = makeNode(
                                        "RecordDef",
                                        makeLeaf(
                                                "RecordId",
                                                child(2,<typeDefinition>).entry
                                        ),
                                        <fieldDefinitions>.ptr
                                );
        free(child(1…4, <typeDefinition>));
}
```

25. <typeDefinition> ===> TK_UNION TK_RUID <fieldDefinitions> TK_ENDUNION

```
{
        <typeDefinition>.ptr = makeNode("UnionDef",
                                        makeLeaf(
                                                "UnionId",
                                                child(2,<typeDefinition>).entry
                                        ),
                                        <fieldDefinitions>.ptr
                                );
        free(child(1…4, <typeDefinition>));
}
```

26. <fieldDefinitions> ===> <fieldDefinition>$_1$ <fieldDefinition>$_2$ <moreFields>

```
{
        <fieldDefinitions>.ptr = makeLinkedListNode("FieldDefLinkedListNode");
        <fieldDefinitions>.ptr.data = <fieldDefinition>1.ptr;
        <fieldDefinitions>.ptr.next = makeLinkedListNode(
                                        "FieldDefLinkedListNode"
                                 );
        <fieldDefinitions>.ptr.next.data =  <fieldDefinition>2.ptr;
        <fieldDefinitions>.ptr.next.next = <moreFields>.ptr;
        free(child(1…3, <fieldDefinitions>));
}
```

27. <fieldDefinition> ===> TK_TYPE <fieldType> TK_COLON TK_FIELDID TK_SEM

```
{
        <fieldDefinition>.ptr = makeNode(
                                        "FieldDef",
                                        <fieldType>.ptr,
                                        child(4, <fieldDefinition>).entry)
                                );
        free(child(1…5, <fieldDefinition>));
}
```

28. <fieldType> ===> <primitiveDatatype>

```
{
        <fieldType>.ptr = <primitiveDatatype>.ptr;
        free(child(1, <fieldType>));
}
```

29. <fieldType> ===> TK_RUID

```
{
        <fieldType>.ptr = makeLeaf(
                                "RecUnionId",
                                child(1, <fieldType>).entry
                        );
        free(child(1, <fieldType>));
}
```

30. <moreFields> ===> <fieldDefinition> <moreFields>$_1$

```
{
        <moreFields>.ptr = makeLinkedListNode("FieldDefLinkedListNode");
        <moreFields>.ptr.data = <fieldDefinition>.ptr;
        <moreFields>.ptr.next = <moreFields>1.ptr;
        free(child(1... 2, <moreFields>));
}
```

31. <moreFields> ===> epsilon

```
{
        <moreFields>.ptr = NULL;
}
```

32. <declarations> ===> <declaration> <declarations>$_1$

```
{
        <declarations>.ptr = makeLinkedListNode("DeclLinkedListNode");
        <declarations>.ptr.data = <declaration>.ptr;
        <declarations>.ptr.next = <declarations>1.ptr;
        free(child(1... 2, <declarations>));
}
```

33. <declarations> ===> epsilon

```
{
        <declarations>.ptr = NULL;
}
```

34. <declaration> ===> TK_TYPE <dataType> TK_COLON TK_ID <global_or_not> TK_SEM

```
{
        declaration.ptr = makeNode(
                                "Declaration",
                                <dataType>.ptr,
                                makeLeaf(TK_ID, TK_ID.entry),
                                global_or_not.ptr
                        );
        free(child(1…6, <declaration>));
}
```

35. <global_or_not> ===> TK_COLON TK_GLOBAL

```
{
        <global_or_not>.ptr = makeLeaf("Global");
        free(child(1, <global_or_not>));
}
```

36. <global_or_not> ===> epsilon

```
{
        <global_or_not>.ptr  = NULL;
}
```

37. <otherStmts> ===> <stmt> <otherStmts>$_1$

{

<otherStmts>.ptr = makeLinkedListNode("*StmlLinkedListNode*");

<otherStmts>.ptr.data = <stmt>.ptr;

<otherStmts>.ptr.next = <otherStmts>$_1$.ptr;

free(child(1…2, <otherStmts>));

}

38. <otherStmts> ===> epsilon

{

<otherStmts>.ptr = NULL;

}

39. <stmt> ===> <assignmentStmt>

{

<stmt>.ptr = <assignmentStmt>.ptr;

free(child(1, <stmt>));

}

40. <stmt> ===> <iterativeStmt>

{

<stmt>.ptr = <iterativeStmt>.ptr;

free(child(1, <stmt>));

}

41. <stmt> ===> <conditionalStmt>

{

<stmt>.ptr = <conditionalStmt>.ptr;

free(child(1, <stmt>));

}

42. <stmt> ===> <ioStmt>

{

<stmt>.ptr = <ioStmt>.ptr;

free(child(1, <stmt>));

}

43. <stmt> ===> <funCallStmt>

```
{
        <stmt>.ptr = <funCallStmt>.ptr;
        free(child(1, <stmt>));
}
```

44. <assignmentStmt> ===> <singleOrRecId> TK_ASSIGNOP <arithmeticExpression> TK_SEM

```
{
        <assignmentStmt>.ptr = makeNode(
                                        "op_ASSIGN",
                                        <singleOrRecId>.ptr,
                                        <arithmeticExpression>.ptr
                                );
        free(child(1…4, <assignmentStmt>));
}
```

45. <singleOrRecId> ===> TK_ID <optionSingleConstructed>

```
{
        <singleOrRecId>.ptr = makeLinkedListNode(
                                        "SingleOrRecIdLinkedListNode"
                                );
        <singleOrRecId>.ptr.data = makeLeaf("Id",
                                        child(1, <singleOrRecId>).entry
                                );
        <singleOrRecId>.ptr.next =  <optionSingleConstructed>.ptr;
        free(child(1…2, <idList>));
}
```

46. <optionSingleConstructed> ===> <oneExpansion> <moreExpansions>

```
{
        <optionSingleConstructed>.ptr = makeLinkedListNode(
                                        "FieldIdLinkedListNode"
                                );
        <optionSingleConstructed>.ptr.data = <oneExpansion>.ptr;
        <optionSingleConstructed>.ptr.next = <moreExpansions>.ptr;
        free(child(1…2, <option_single_constructed>));
}
```

47. <optionSingleConstructed> ===> epsilon

```
{
        <optionSingleConstructed>.ptr = NULL;
}
```

48. <oneExpansion> ===> TK_DOT TK_FIELDID

```
{
       <oneExpansion>.ptr =makeLeaf("FieldId", child(2,<oneExpansion>).entry);
}
```

49. <moreExpansions> ===> <oneExpansion> <moreExpansions>

```
{
        <moreExpansions>.ptr = makeLinkedListNode("FieldIdLinkedListNode");
        <moreExpansions>.ptr.data = <oneExpansion>.ptr;
        <moreExpansions>.ptr.next = <moreExpansions>.ptr;
        free(child(1…2, <moreExpansions>));
}
```

50. <moreExpansions> ===> epsilon

```
{
        <moreExpansions>.ptr = NULL;
}
```

51. <funCallStmt> ===> <outputParameters> TK_CALL TK_FUNID TK_WITH TK_PARAMETERS <inputParameters> TK_SEM

```
{
        <funCallStmt> = makeNode(
                            "FuncCall",
                            outputParameters.ptr,
                            makeLeaf("FuncId", child(3, <funCallStmt>).entry),
                            inputParameters.ptr
                        );
        free(child(1…7, <funCallStmt>));
}
```

52. <outputParameters> ===> TK_SQL <idList> TK_SQR TK_ASSIGNOP

```
{
        <outputParameters>.ptr = <idList>.ptr;
        free(child(1…4, <outputParameters>));
```

}

53. <outputParameters> ===> epsilon

{

<outputParameters>.ptr = NULL;

}

54. <inputParameters> ===> TK_SQL <idList> TK_SQR

{

<inputParameters>.ptr = <idList>.ptr;

free(child(1…3, <inputParameters>));

}

55. <iterativeStmt> ===> TK_WHILE TK_OP <booleanExpression> TK_CL <stmt> <otherStmts> TK_ENDWHILE

{

temp = makeLinkedListNode("*StmtLinkedListNode*");

temp.data = <stmt>.ptr;

temp.next = <otherStmts>.ptr;

<iterativeStmt>.ptr = makeNode(

"*While*",

<booleanExpression>.ptr,

temp

);

free(child(1…7, <iterativeStmt>));

}

56. <conditionalStmt> ===> TK_IF TK_OP <booleanExpression> TK_CL TK_THEN <stmt> <otherStmts> <elsePart>

Conditional statement's required values:

- **Relational expression**
- **Statements inside if**
- **Statements inside else**

{

temp = makeLinkedListNode("*StmtLinkedListNode*");

temp.data = <stmt>.ptr;

temp.next = <otherStmts>.ptr;

```
        <conditionalStmt>.ptr = makeNode(
                                        "If",
                                        <booleanExpression>.ptr,
                                        temp,
                                        <elsePart>.ptr
                                    );
        free(child(1…8, <conditionalStmt>));
    }
```

57. <elsePart> ===> TK_ELSE <stmt> <otherStmts> TK_ENDIF

```
    {
        temp = makeLinkedListNode("StmtLinkedListNode");
        temp.data = <stmt>.ptr;
        temp.next = <otherStmts>.ptr;
        <elsePart>.ptr = makeNode("Else",temp);
        free(child(1…4, <elsePart>));
    }
```

58. <elsePart> ===> TK_ENDIF

```
    {
        <elsePart>.ptr = NULL;
        free(child(1, <elsePart>));
    }
```

59. <ioStmt> ===> TK_READ TK_OP <var> TK_CL TK_SEM

```
    {
        <ioStmt>.ptr = makeNode("Read", <var>.ptr);
        free(child(1…5, <ioStmt>));
    }
```

60. <ioStmt> ===> TK_WRITE TK_OP <var> TK_CL TK_SEM

```
    {
        <ioStmt>.ptr = makeNode("Write", <var>.ptr);
        free(child(1…5, <ioStmt>));
    }
```

61. <arithmeticExpression> ===> <term> <expPrime>

```
{
        <expPrime>.inhptr = <term>.ptr;  // this action needs to be performed while
                                            moving from traversing <term> to
                                            <expPrime>

        <arithmeticExpression>.ptr = <expPrime>.ptr;  // this action is performed
                                                         after traversing both the
                                                         children completely.
        free(child(1...2,<arithmeticExpression>));
}
```

62. <expPrime> ===> <lowPrecedenceOperators> <term> <expPrime>$_1$

```
{
        // the below action is performed after traversing the <term> child in RHS of
        the production rule, before traversing <expPrime1>
        <expPrime>1.inhptr = makeNode(
                                    <lowPrecendenceOperators>.ptr.name,
                                    <expPrime>.inhptr,
                                    <lowPrecedenceOperators>.ptr,
                                    <term>.ptr
                                );

        //these actions are taken after traversing all children of <expPrime>
        <expPrime>.ptr = <expPrime1>.ptr;
        free(child(1...3,<expPrime>));
}
```

63. <expPrime> ===> epsilon

```
{
        <expPrime>.ptr = <expPrime>.inhptr;
}
```

64. <term> ===> <factor> <termPrime>

The value stored in factor needs to be transferred down to termPrime

```
{
        <termPrime>.inhptr = <factor>.ptr;  // this action needs to be performed
                                              while  moving from traversing <factor>
                                              to <termPrime>
        <term>.ptr = <termPrime>.ptr; // this action is performed after traversing
                                        both the children completely.
        free(child(1...2,<term>));
}
```

65. <termPrime> ===> <highPrecedenceOperators> <factor> <termPrime>$_1$

```
{
        <termPrime>1.inhptr = makeNode(
                                    <highPrecendenceOperators>.ptr.name,
                                    <termPrime>.inhptr,
                                    <highPrecedenceOperators>.ptr,
                                    <factor>.ptr
                                ); //this action is performed after traversing the <term>
                                   child in RHS of the production rule, before traversing
        <termPrime>.ptr = <termPrime1>.ptr; //these actions are taken after
                                              traversing all children of
                                              <expPrime>
        free(child(1...3,<termPrime>));
}
```

66. <termPrime> ===> epsilon

```
{
        <termPrime>.ptr = <termPrime>.inhptr;
}
```

67. <factor> ===> TK_OP <arithmeticExpression> TK_CL

{
<factor>.ptr = <arithmeticExpression>.ptr;
free(child(1…3, <factor>));
}

68. <factor> ===> <var>

{
<factor>.ptr = <var>.ptr;
free(child(1, <factor>));
}

69. <highPrecedenceOperators> ===> TK_MUL

{
<highPrecedenceOperators>.ptr = makeLeaf("*arithOp_MUL*");
free(child(1, <highPrecedenceOperators>));
}

70. <highPrecedenceOperators> ===> TK_DIV

{
<highPrecedenceOperators>.ptr = makeLeaf("*arithOp_DIV*");
free(child(1, <highPrecedenceOperators>));
}

71. <lowPrecedenceOperators> ===> TK_PLUS

{
<lowPrecedenceOperators>.ptr = makeLeaf("*arithOp_PLUS*");
free(child(1, <lowPrecedenceOperators>));
}

72. <lowPrecedenceOperators> ===> TK_MINUS

```
{
        <lowPrecedenceOperators>.ptr = makeLeaf("arithOp_MINUS");
        free(child(1, <lowPrecedenceOperators>));
}
```

73. <booleanExpression> ===> TK_NOT TK_OP <booleanExpression> TK_CL

```
{
        <booleanExpression>.ptr = makeNode(
                                        "logOp_NOT",
                                        <booleanExpression>1.ptr
                                );
        free(child(1…4, <booleanExpression>));
}
```

74. <booleanExpression> ===> TK_OP <booleanExpression> TK_CL <logicalOp> TK_OP <booleanExpression> TK_CL

```
{
        <booleanExpression>.ptr = makeNode(
                                        <logicalOp>.ptr.name,
                                        <booleanExpression>1.ptr,
                                        <booleanExpression>2.ptr
                                );
        free(child(1…7, <booleanExpression>));
}
```

75. <booleanExpression> ===> <var> <relationalOp> <var>

```
{
        <booleanExpression>.ptr = makeNode(
                                        <relationalOp>.ptr.name,
                                        <var>1.ptr,
                                        <var>2.ptr
                                );
        free(child(1…3, <booleanExpression>));
}
```

76. <var> ===> <singleOrRecId>

```
{
        <var>.ptr = <singleOrRecId>.ptr;
        free(child(1, <var>));
}
```

77. <var> ===> TK_NUM

```
{
        <var>.ptr = makeLeaf("Num", TK_NUM.value);
        free(child(1, <var>));
}
```

78. <var> ===> TK_RNUM

```
{
        <var>.ptr = makeLeaf("RealNum", TK_RNUM.value);
        free(child(1, <var>));
}
```

79. <logicalOp> ===> TK_AND

```
{
        <logicalOp>.ptr = makeLeaf("logOp_AND");
        free(child(1, <logicalOp>));
}
```

80. <logicalOp> ===> TK_OR

```
{
        <logicalOp>.ptr = makeLeaf("logOp_OR");
        free(child(1, <logicalOp>));
}
```

81. <relationalOp> ===> TK_LT

```
{
        <relationalOp>.ptr = makeLeaf("relOp_LT");
        free(child(1, <logicalOp>));
}
```

82. **<relationalOp> ===> TK_LE**

```
{
        <relationalOp>.ptr = makeLeaf("relOp_LE");
        free(child(1, <relationalOp>));
}
```

83. **<relationalOp> ===> TK_EQ**

```
{
        <relationalOp>.ptr = makeLeaf("relOp_EQ");
        free(child(1, <relationalOp>));
}
```

84. **<relationalOp> ===> TK_GT**

```
{
        <relationalOp>.ptr = makeLeaf("relOp_GE");
        free(child(1, <relationalOp>));
}
```

85. **<relationalOp> ===> TK_GE**

```
{
        <relationalOp>.ptr = makeLeaf("relOp_GE")
        free(child(1, <relationalOp>));
}
```

86. **<relationalOp> ===> TK_NE**

```
{
        <relationalOp>.ptr = makeLeaf("relOp_NE");
        free(child(1, <relationalOp>));
}
```

87. **<returnStmt> ===> TK_RETURN <optionalReturn> TK_SEM**

```
{
        <returnStmt>.ptr = makeNode("Return", <optionalReturn>.ptr);
        free(child(1…3, <returnStmt>));
}
```

88. **<optionalReturn> ===> TK_SQL <idList> TK_SQR**

```
{
        <optionalReturn>.ptr = <idList>.ptr;
        free(child(1…3, <optionalReturn>));
}
```

89. **<optionalReturn> ===> epsilon**

```
{
        <optionalReturn>.ptr = NULL;
}
```

90. **<idList> ===> TK_ID <more_ids>**

```
{
        <idList>.ptr = makeLinkedListNode("IdLinkedListNode");
        <idList>.ptr.data = makeLeaf("Id", child(1, <idList>).entry);
        <idList>.ptr.next =  <more_ids>.ptr;
        free(child(1…2, <idList>));
}
```

91. **<more_ids> ===> TK_COMMA <idList>**

```
{
        <more_ids>.ptr = <idList>.ptr;
        free(child(1…2, <more_ids>));
}
```

92. **<more_ids> ===> epsilon**

```
{
        <more_ids>.ptr = NULL;
}
```

93. **<definetypestmt> ===> TK_DEFINETYPE <A> TK_RUID TK_AS TK_RUID**

```
{
        <definetypestmt>.ptr = makeNode(
                                        "DefineType",
                                        <A>.structType,
                                        makeLeaf(
                                                "RecUnionId",
                                                child(3, <definetypestmt>).entry
                                        ),
                                        makeLeaf(
```

```
                                                    “RecUnionId”,
                                                    child(5, <definetypestmt>).entry
                                            )
                                    );
        free(child(1…5, <A>));
    }
```

94. <A> ===> TK_RECORD

```
    {
        <A>.structType = makeLeaf(“Record”);
        free(child(1, <A>));
    }
```

95. <A> ===> TK_UNION

```
    {
        <A>.structType = makeLeaf(“Union”);
        free(child(1, <A>));
    }
```