编译器构造实验(2): 语法分析器

Mar 30, 2021

1. Syntax Analyzer

In this phase of the project, you are required to write a syntax parser using YACC for the programming language, MINI-JAVA. The parser communicates with the lexer you built in Project 1 and outputs the parse tree of the input MINI-JAVA program.

2. Due Date [截止时间]

The project is due **Apr 15, 2021**.

3. Grammar Specification［文法规则］

The grammar is specified by syntax diagrams given in Appendix B.

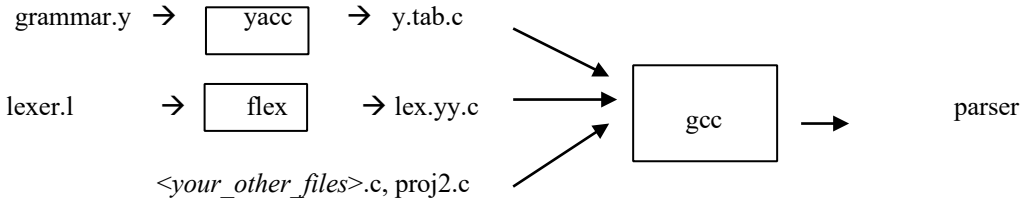
A syntax diagram is a directed graph with one entry and one exit. Each path through the graph defines an allowable sequence of symbols. For example, the structure of a valid MINI-JAVA program is defined by the first syntax diagram, which means that the program begins with 'program <name>;' and consists of zero or multiple classes.

4. Implementation［具体实现］

Overall, for this project, you should complete the following tasks:

a. develop a context free grammar (CFG) for the MINI-JAVA from the syntax diagrams;
b. express the developed grammar as a YACC specification;
c. extend your YACC code to construct a parser to accept a syntactically correct MINI-JAVA program and report errors on incorrectly structured programs, and also generate the syntax tree for the source program.

4.1 Parser Structure



terminal# **parser < test1.java**

"grammar.y" has similar file structure as that of "lexer.l".

```
%{ /* definition */
#include  "proj2.h"
#include  <stdio.h>
%}
%token <intg>  PROGRAMnum IDnum ....  SCONSTnum
%type  <tptr>  Program ClassDecl   .....  Variable

%%  /* yacc specification */
Program  :  PROGRAMnum IDnum SEMInum ClassDecl
            {  $$ = MakeTree(ProgramOp, $4, MakeLeaf(IDNode, $2)); printtree($$, 0);  }
            ;
/* other rules */
Expression  :   SimpleExpression   {$$ = $1;}
                |  SimpleExpression Comp_op SimpleExpression
                    {  MkLeftC($1, $2);   $$ = MkRightC($3, $2);   }

%%
int yycolumn, yyline;
FILE *treelst;
main()   {   treelst = stdout;   yyparse(); }
yyerror(char *str)  {   printf("yyerror: %s at line %d\n", str, yyline); }
```

Some modifications have to be made in your lexer.l from your Project 1. In all the places you assign yylval, you need to assign yylval.intg instead as such:

```
{int}            {yycolumn += yyleng; yylval.intg = atoi(yytext); return(ICONSTnum);}
{variable}       { .... yylval.intg = index; ... }
```

This is because yylval is now declared as a union to accommodate both token values and tree nodes.

4.2 Data Structures

Appendix A lists functions that are provided for your convenience to implement and debug your code. The C source code “proj2.c” and header file “proj2.h” are provided for you to use. Inside proj2.h, a tree node is declared as such:

```
typedef struct treenode {
        int NodeKind, NodeOpType, IntVal; struct
        treenode *LeftC, *RightC;
} ILTree, *tree;
```

The NodeKind field distinguishes between the following types of nodes: **IDNode**, **NUMNode**, **STRINGNode**, **DUMMYNode**, **INTEGERTNode** or **EXPRNode**. The first four leaf node types correspond to an identifier, an integer constant, a string constant and a null node type. A leaf node of INTEGERTNode type is created for “int” type declarations, i.e. the node is created for every INTnum token. All interior nodes are of the EXPRNode type.

Each leaf node assigns the IntVal field. For an ID or string constant node, IntVal is the index into the string table. For a NUMNode, it is the value itself. For an INTEGERTNode or DUMMYNode, it is always 0.

Each interior node assign the NodeOpType field, the values of which are defined in proj2.h:

ProgramOp:	program, root node operator
BodyOp:	class body, method body, decl body, statmentlist body.
DeclOp:	each declaration has this operator
CommaOp:	connected by ","
ArrayTypeOp:	array type
TypeIdOp:	type id operator
BoundOp:	bound for array variable declaration
HeadOp:	head of method,
RArgTypeOp:	arguments
VargTypeOp:	arguments specified by "VAL", e.g., abc(VAL int x)
StmtOp:	statement
IfElseOp:	if-then-else
LoopOp:	while statement
SpecOp:	specification of parameters
RoutineCallOp:	routine call
AssignOp:	assign operator
ReturnOp:	return statement
AddOp, SubOp, MultOp, DivOp, LTOp, GTOp, EQOp, NEOp, LEOp, GEOp, AndOp, OrOp, UnaryNegOp, NotOp:	ALU operators
VarOp:	variables
SelectOp:	to access a field/index variable
IndexOp:	follow "[]" to access a variable
FieldOp:	follow "." To access a variable
ClassOp:	for each class
MethodOp:	for each method
ClassDefOp:	for each class definition

Functions MakeLeaf, MakeTree are used to create leaf nodes and intermediate nodes respectively. printtree(tree nd, int depth) is used to output a tree structure. You need to provide the implementation of the following two functions in order to have variable name and string const correctly printed. That is, replace the following code in "proj2.c" with your version:

```
extern char  strg_tbl[];

char* getname(int i) /* i is the index of the  table, passed through yylval*/
        { return( strg_tbl+i );/*return string table indexed at i*/ }

char* getstring(int i)
        { return( strg_tbl+i );/*return string table indexed at i*/}
```

In addition, you are also required to print out the parse tree from the top after you have successfully built it. Syntax errors should be reported in your yyerror function. You need to give the line number where an error occurs.

```
/* Example 1: A hello world program */
program xyz;
class Test {
   method void main() {
      System.println('Hello World !!!');
   }
}
```

A sample output for the Hello World example (as shown in the above table) given in Project 1 is:

```
  +-[IDNode,0,"xyz"]
R-[ProgramOp]
  |   +-[IDNode,4,"Test"]
  | +-[ClassDefOp]
  | | |           +-[DUMMYnode]
  | | |         +-[CommaOp]
  | | |         | +-[STRINGNode,29,"Hello World !!!"]
  | | |       +-[RoutineCallOp]
  | | |       | |   +-[DUMMYnode]
  | | |       | | +-[SelectOp]
  | | |       | | | | +-[DUMMYnode]
  | | |       | | | +-[FieldOp]
  | | |       | | |   +-[IDNode,21,"println"]
  | | |       | +-[VarOp]
  | | |       |   +-[IDNode,14,"System"]
  | | |     +-[StmtOp]
  | | |     | +-[DUMMYnode]
  | | |   +-[BodyOp]
  | | |   | +-[DUMMYnode]
  | | | +-[MethodOp]
  | | | | |   +-[DUMMYnode]
  | | | | | +-[SpecOp]
  | | | | | | +-[DUMMYnode]
  | | | | +-[HeadOp]
  | | | |   +-[IDNode,9,"main"]
  | | +-[BodyOp]
  | |   +-[DUMMYnode]
  +-[ClassOp]
     +-[DUMMYnode]
```

The tree is printed starting at the root node. The provided routines traverse the nodes of the tree in a inorder traversal printing the right subtree before the left subtree (i.e., Right → Root → Left). Each node is printed on a separate line. The root node is printed in the first column, and all nodes at the same level are printed with the same indentation. If level *i* is printed starting in column *j*, then level *i+1* is printed starting at column *j+1*.

5. Assignment Submission [关于提交]

When you are done, create a gzipped tarball of your source files. You **must** include a file that shows how to compile/execute your code named README.txt. Preferably, include a Makefile. The submission should be a compressed file that contains your project source code and README (no executable). Submission details will be provided afterwards in the discussion forum, please stay tuned.

Appendix A: Provided functions

function NullExp(); return *ILTree

Returns a null node with kind=DummyNode and semantic value=0.

function MakeLeaf(Kind: NodeKindType; N: integer); return *ILTree

Returns a leaf node of specified Kind with integer semantic value N.

function MakeTree(Op: NodeOpType; Left,Right: *ILTree); return *ILTree

Returns an internal node, T, such that NodeOp(T)=Op; LeftChild(T)=Left; RightChild(T)=Right and NodeKind(T)=InteriorNode.

function NodeOp(T: *ILTree); return NodeOpType

See MakeTree. Returns the integer constant representing NodeOpType of T if T is an interior node, else returns UndefOp. Uses NodeKind(T) to distinguish leaf from interior.

function NodeKind(T: *ILTree); return NodeKindType

Returns the kind of node T.

function LeftChild(T: *ILTree); return *ILTree

Returns pointer to left child of T. Returns pointer to null node if NodeKind(T) <> InteriorNode.

function RightChild(T: *ILTree); return *ILTree

Returns pointer to right child of T. Returns pointer to null node if NodeKind(T)!= InteriorNode.

function IntVal(T: *ILTree); return integer

See MakeLeaf. Returns integer semantic value of node T if NodeKind(T) = IDNode, STRGNode, NUMNode, or BOOLNode. Otherwise returns Undefined.

function IsNull(T: *ILTree); return boolean

IsNull(T) iff T is null node.

function SetNodeOp(T: *ILTree; Op: NodeOpType)

NodeKind(T) must be InteriorNode. Makes NodeOp(T) = Op.

function SetNodeKind(T: *ILTree; Kind: NodeKindType)

NodeKind(T) must not be InteriorNode. Makes NodeKind(T) = Kind.

function SetNodeVal(T: *ILTree; Val: integer)

NodeKind(T) must not be InteriorNode. Makes IntVal(T) = Val.

function SetLeftChild(T,NewChild: *ILTree)

NodeKind(T) must be InteriorNode. Makes LeftChild(T) = NewChild.

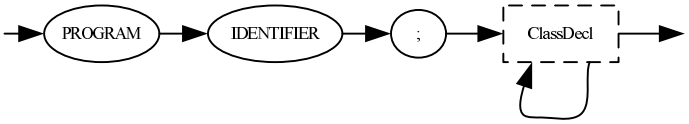
function SetRightChild(T,NewChild: *ILTree)

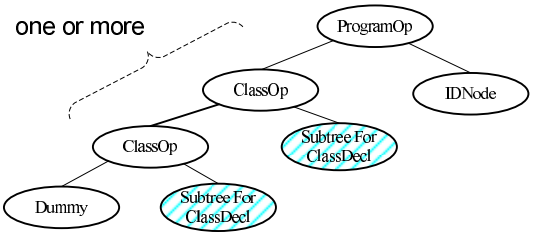
NodeKind(T) must be InteriorNode. Makes RightChild(T) = NewChild.

Appendix B: Syntax diagrams

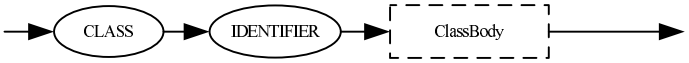
Legend:	dashed boxes → nonterminal symbols solid ellipsis → terminal symbols (tokens)	Legend:	eclipse → normal nodes shaded eclipse → subtree

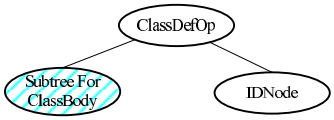
Program



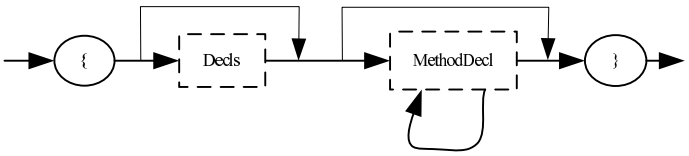


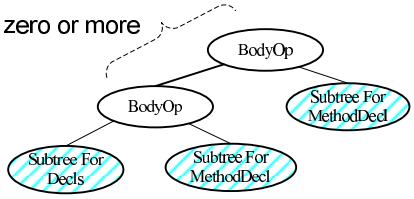
ClassDecl



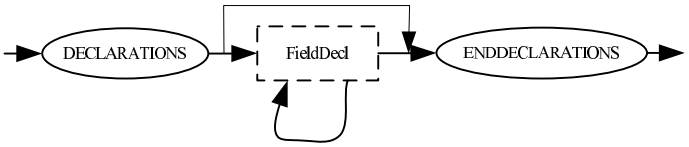


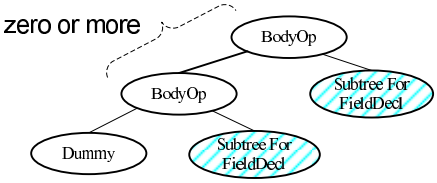
ClassBody



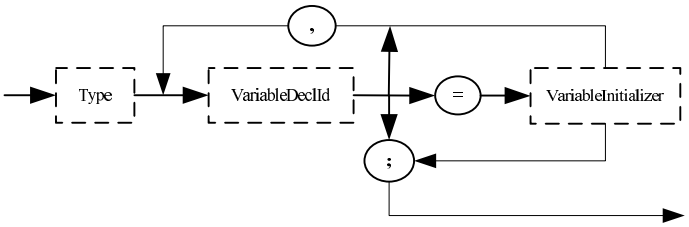


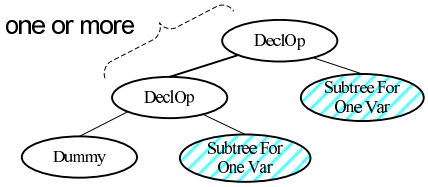
Decls



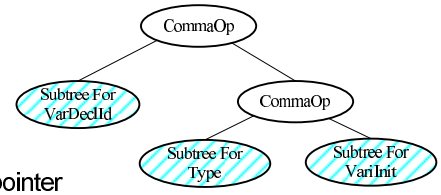


FieldDecl



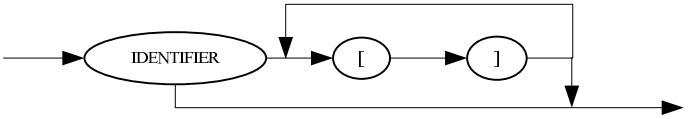


Each Var has the following subtree

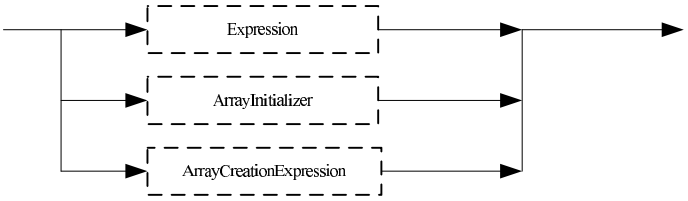


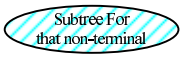
Type should be stored in a separate pointer (global variable) such that it may be used in building the *VariableInitializer* subtree.

VariableDeclId

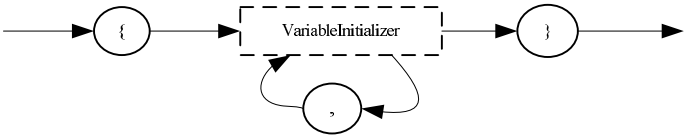


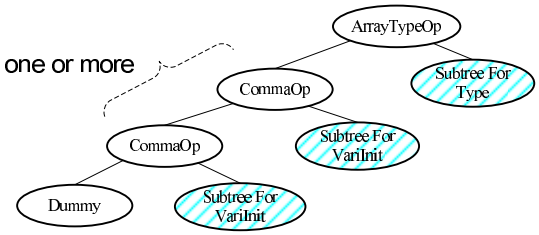
VariableInitializer



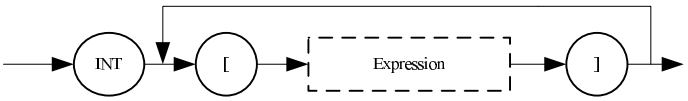


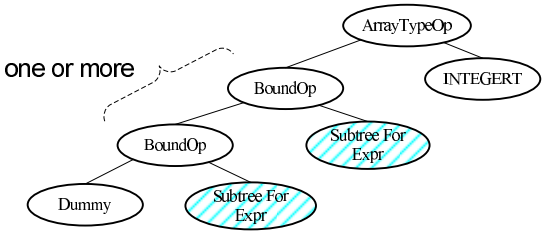
ArrayInitializer



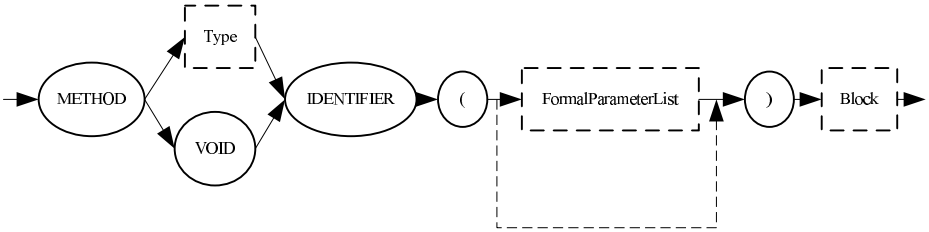


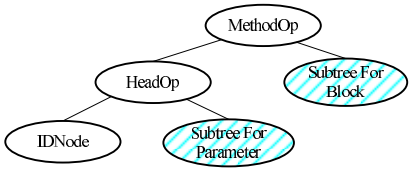
ArrayCreationExpression





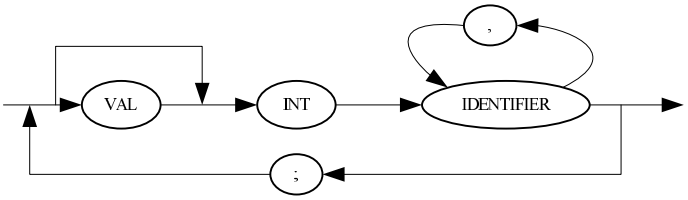
MethodDecl

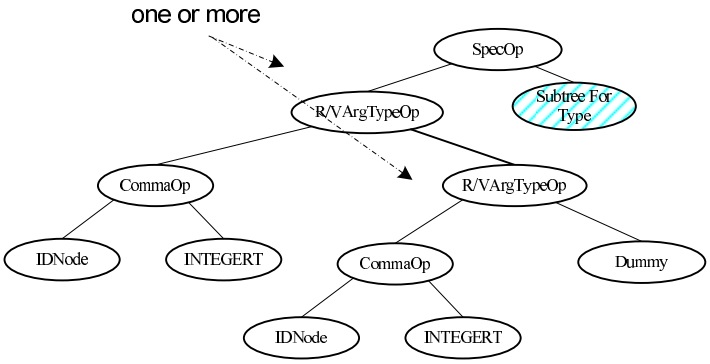




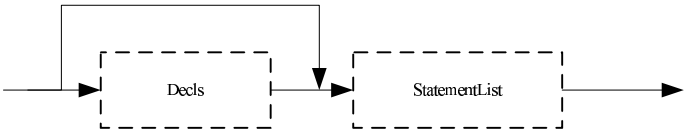
Type should be stored in a separate pointer (global variable) such that it may be used in building the *Parameter* and *Block* subtrees.

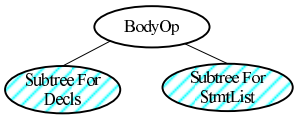
FormalParameterList



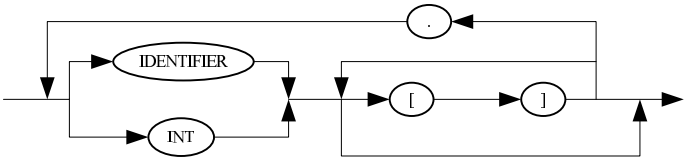


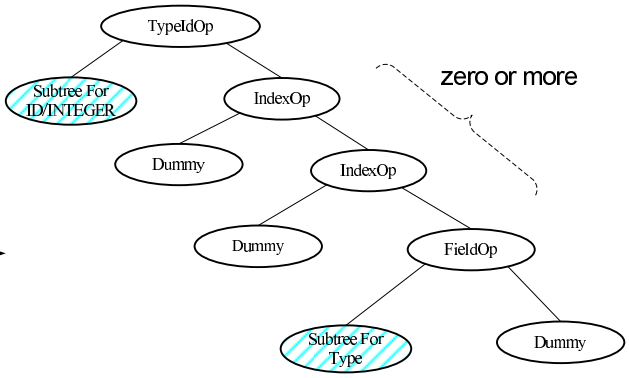
Block



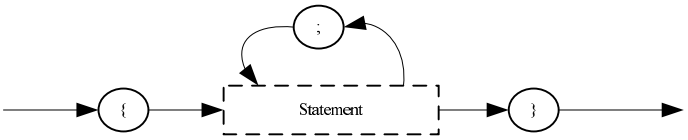


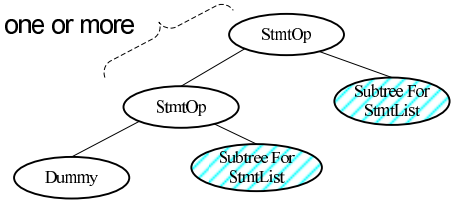
Type



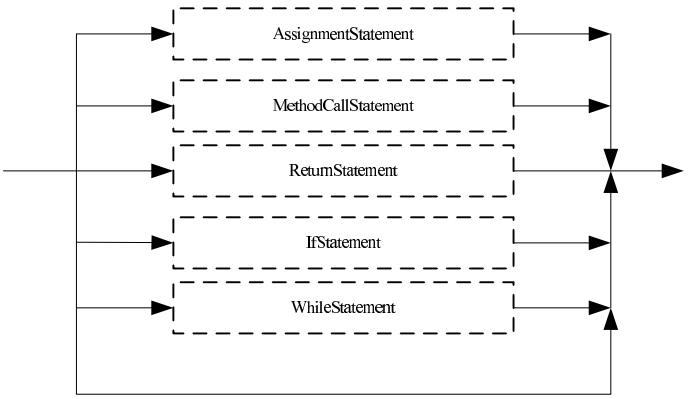


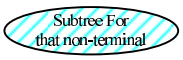
StatementList



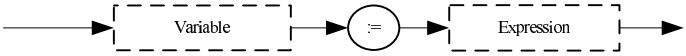


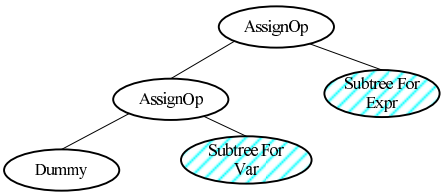
Statement



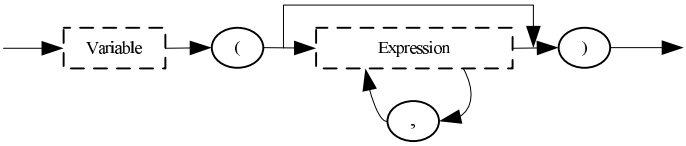


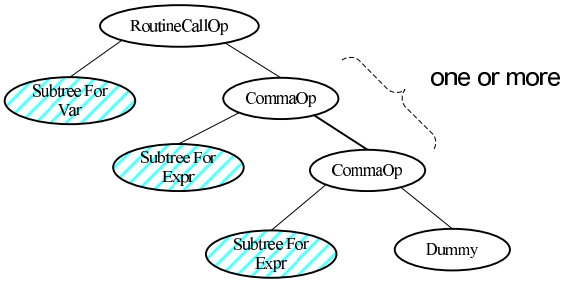
AssignmentStatement



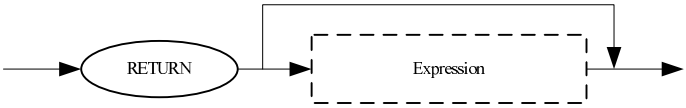


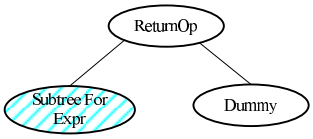
MethodCallStatement



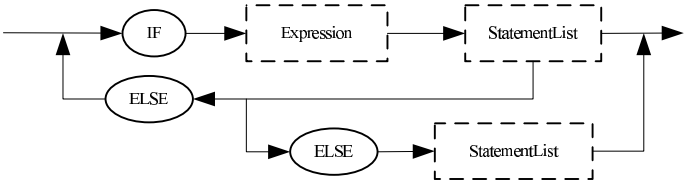


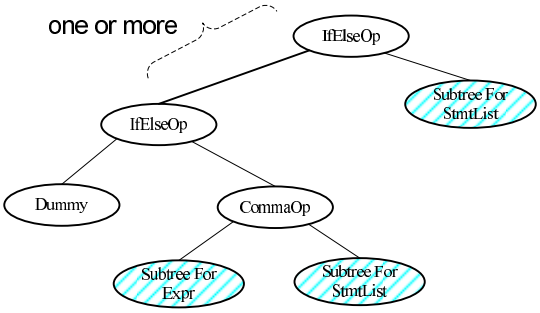
ReturnStatement



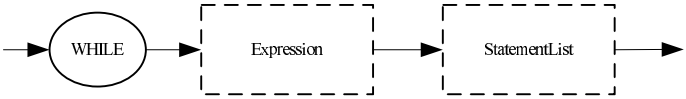


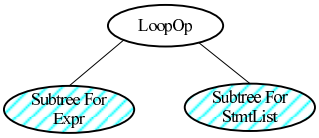
IfStatement



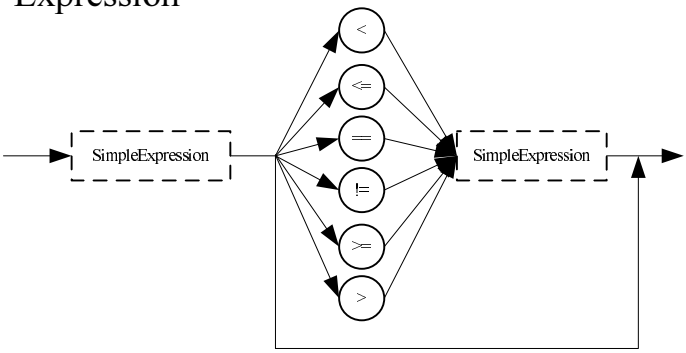


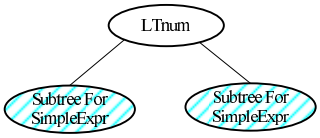
WhileStatement



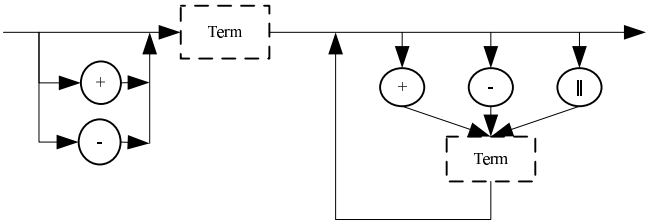


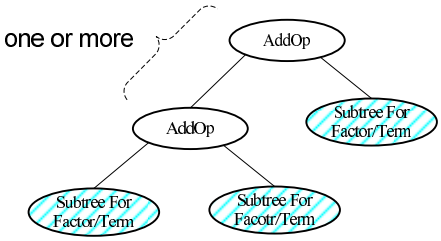
Expression



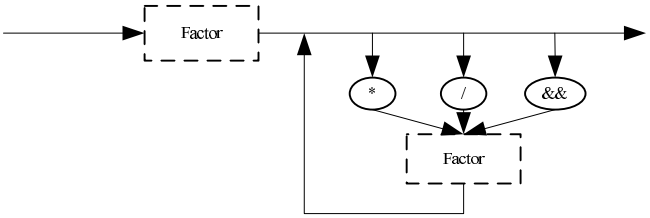


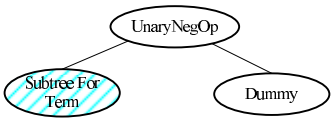
SimpleExpression



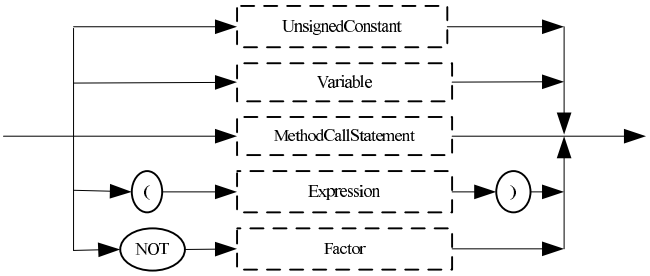


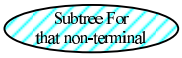
Term



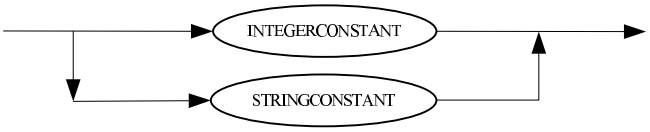


Factor





UnsignedConstant



Variable