CS2210 Compiler Construction

Spring 2019

Part II: Syntax Analysis

1. Objective

In this phase of the project, you are required to write a parser using YACC for the CS 2210 programming language, MINI-JAVA. The parser communicates with the lexer you built in Part I and outputs the parse tree of the input MINI-JAVA program.

2. Due Date

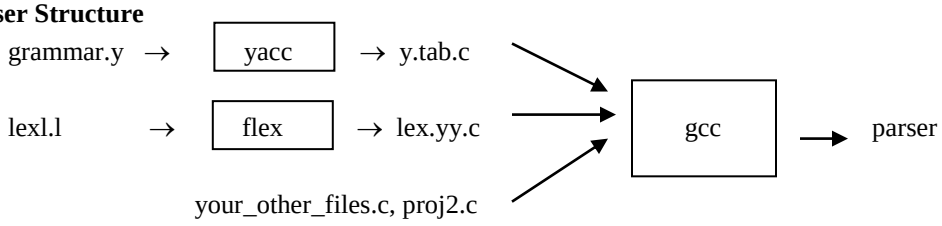
The assignment is due **March 18, 2019** at the beginning of the class.

3. Grammar Specification

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

4. Implementation

4.1 Parser Structure



terminal# **parser < test1.java**

Grammar.y has similar file structure as that of "lex.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); }
#include "lex.yy.c"
```

Some modifications have to be made in your lex.l. 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" could be found from class webpage. 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 defintion

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*/}
```

To grade your project, 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.

A sample output for the Hello World example 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]
```

5. Assignment Submission

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 named Makefile. The submission should be a compressed file that contains your project source code and readme (no executable please). On Linux, this can be done with the command "tar zcvf USERNAME proj2.tar.gz *". Copy your archive to the directory: ~wahn/submit/2210/.

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.

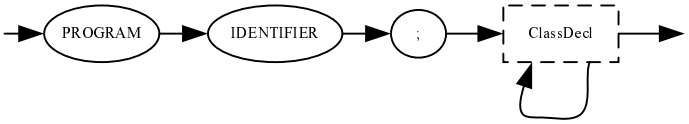
function SetRightChild(T,NewChild: *ILTree)

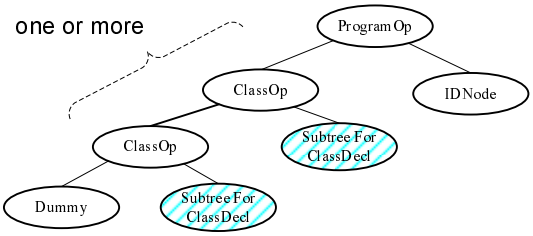
NodeKind(T) must be InteriorNode. Makes RightChild(T) = NewChild.

Appendix B: Syntax diagrams

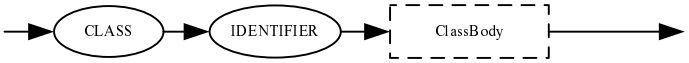
Legend:	dashed boxes → nonterminal symbols solid ellipsis → terminal symbols (tokens)	Legend:	eclipse → normal nodes shaded eclipse → subtree

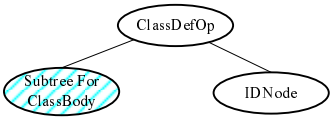
Program



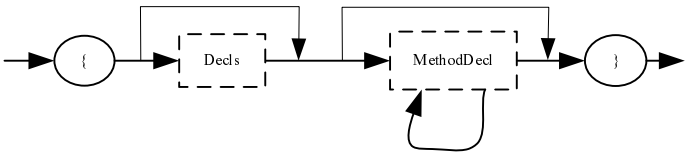


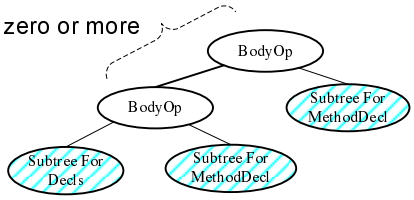
ClassDecl



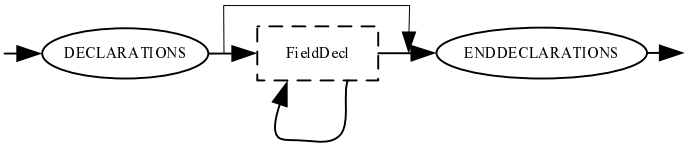


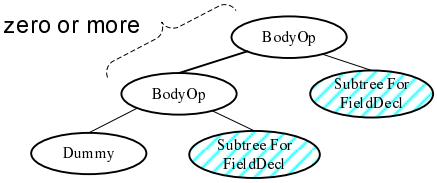
ClassBody



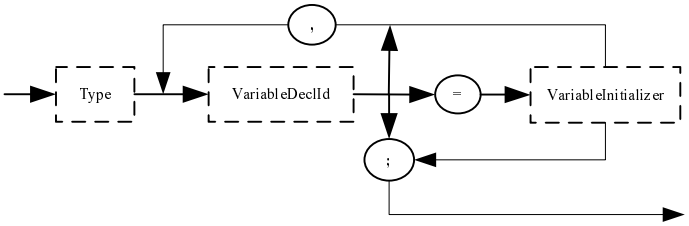


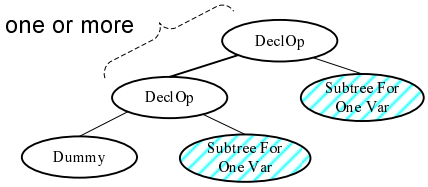
Decls



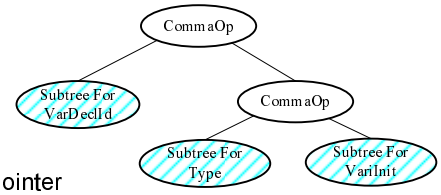


FieldDecl



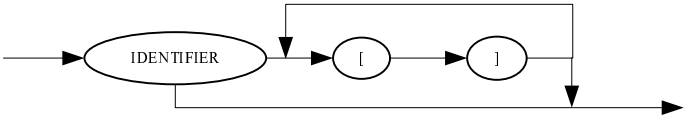


Each Var has the following subtree

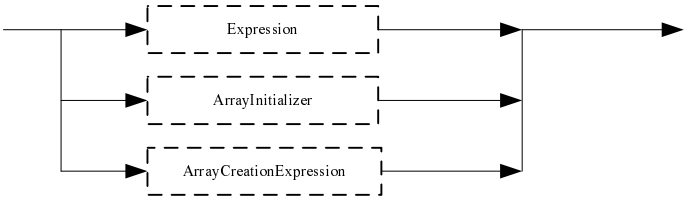


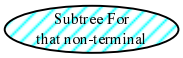
Type should be stored in a separate pointer (global variable) such that it may be used in building the *VariableInitializer* subtree.

VariableDeclId

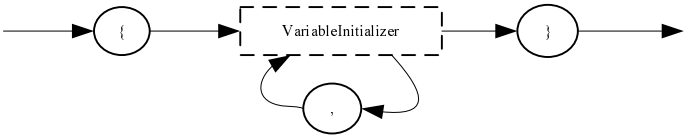


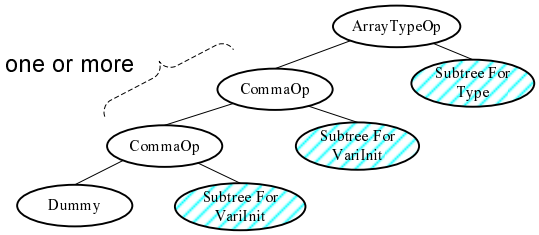
VariableInitializer



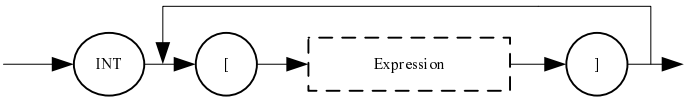


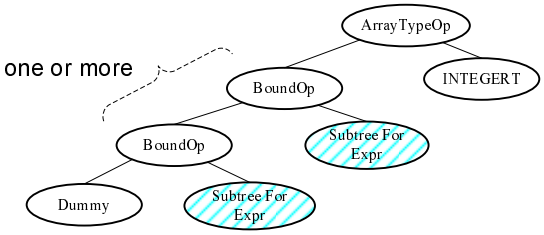
ArrayInitializer



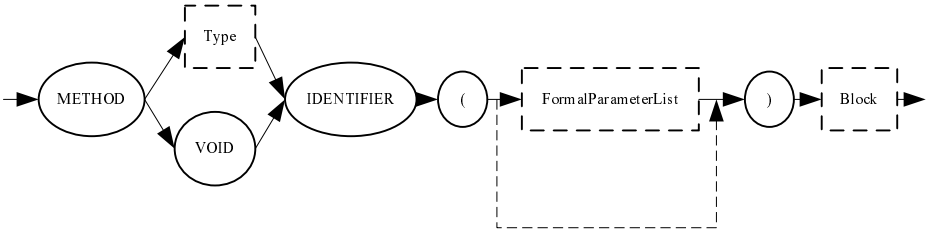


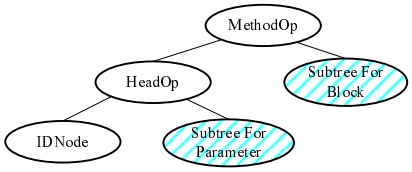
ArrayCreationExpression





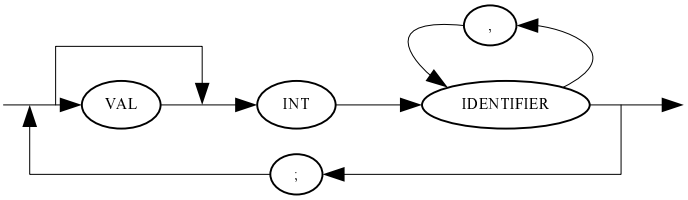
MethodDecl

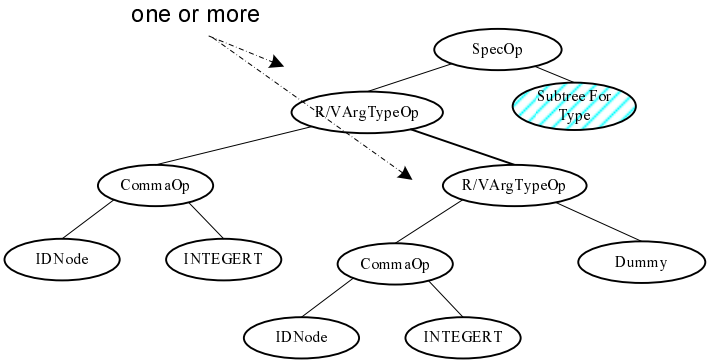




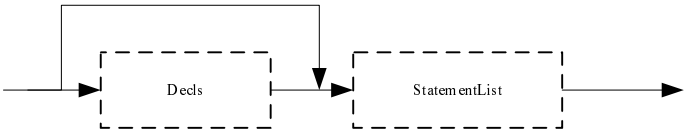
Type should be stored in a separate pointer (global variable) such that it may be used in building the *Parameter* and *Block* subtrees.

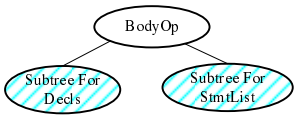
FormalParameterList



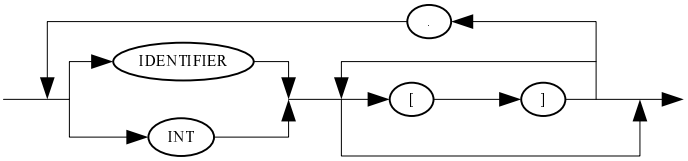


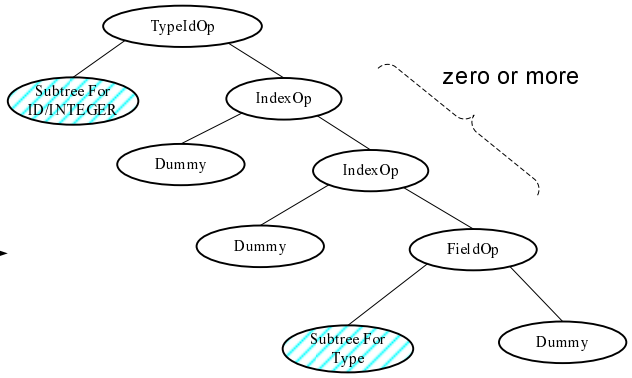
Block



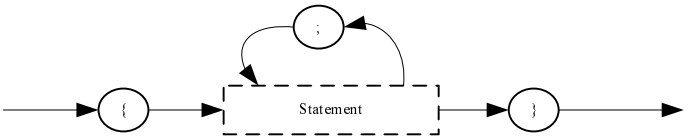


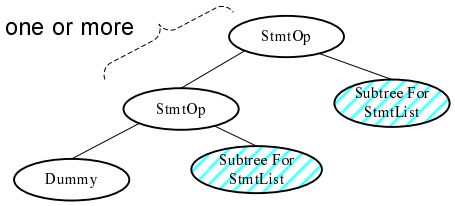
Type



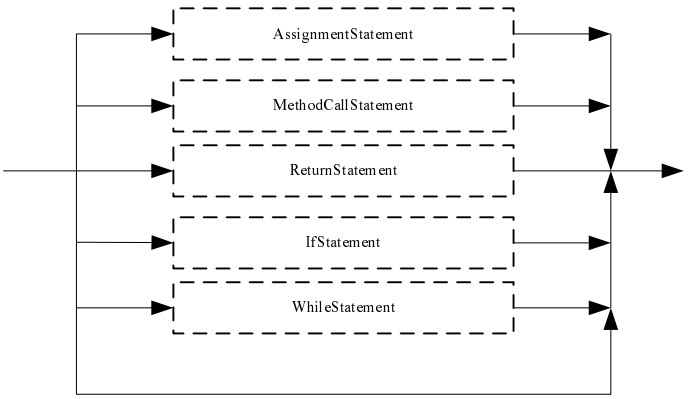


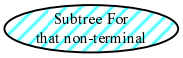
StatementList



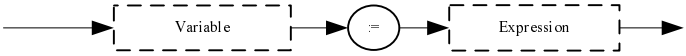


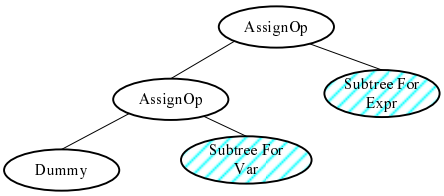
Statement



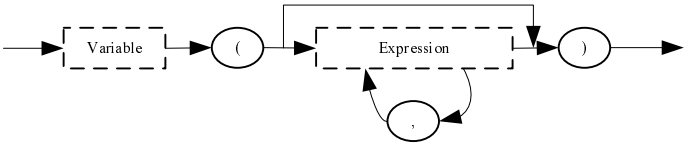


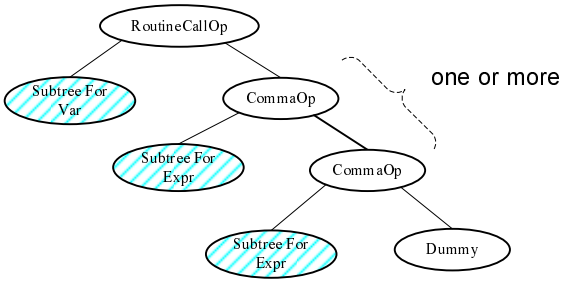
AssignmentStatement



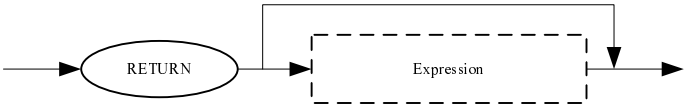


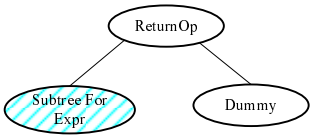
MethodCallStatement



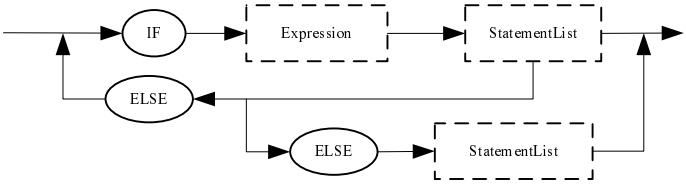


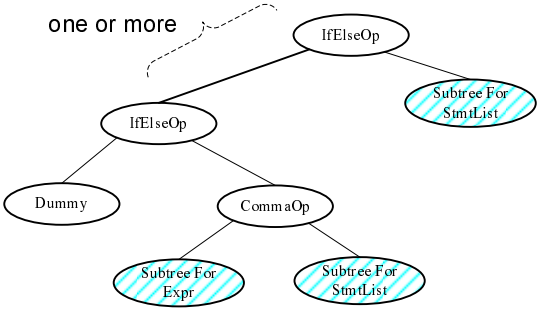
ReturnStatement



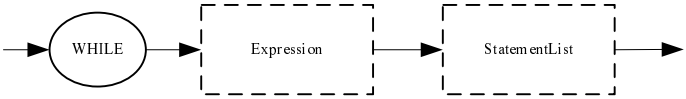


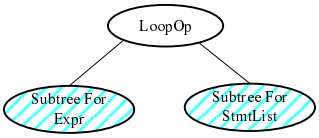
IfStatement



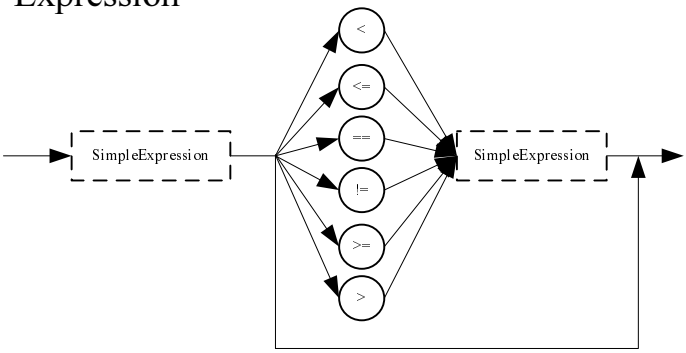


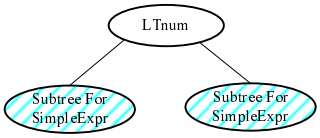
WhileStatement



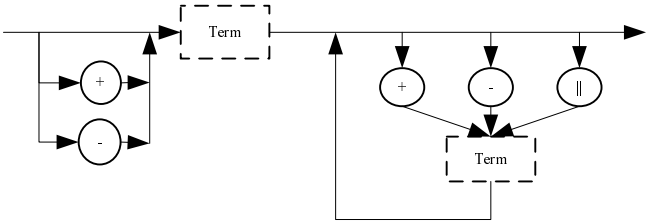


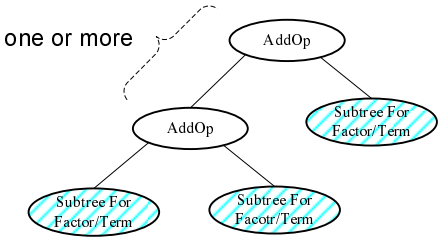
Expression



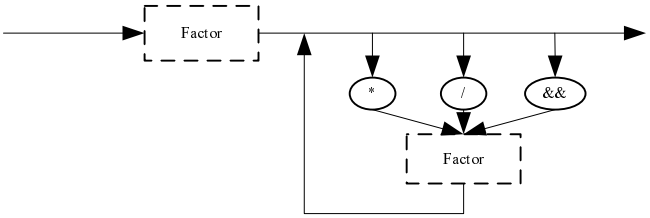


SimpleExpression

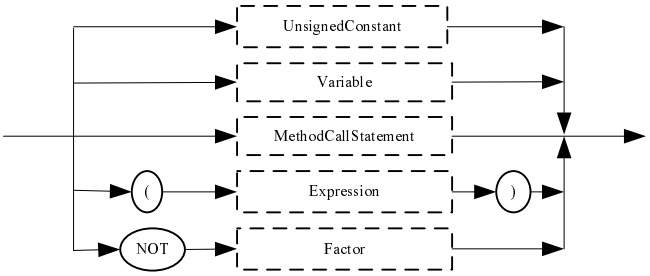


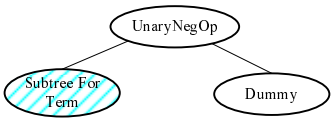


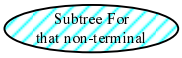
Term



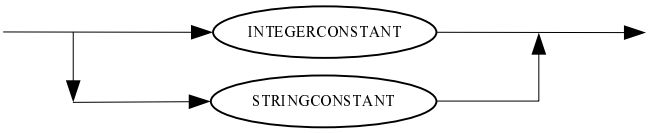
Factor

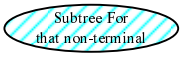






UnsignedConstant





Variable