program -> program\_head program\_body .

program\_head -> PROGRAM ID ( identifier\_list );{cout<<”#include<stdio.h>”<<endl;}

program\_body -> const\_declarations type\_declarations var\_declarations subprogram\_declarations O compound\_statement{cout<<compound\_statement.str<<"}"}

O -> empty{cout<<”main()\n{”;}

identifier\_list -> identifier\_list , ID { identifier\_list.str = identifier\_list1.str + “, ” + ID , item["name"] = p[3] , item["type"] = "" , item["F\_type"] = False , item["F\_const"] = False , item["declare\_line"] = p.lineno(3) , item["extend"] = [] , isDefined = symboltable.find(ID, mode=1); if(!isRecord && !isDefined){ symboltable.insert(item) } else if(isRecord && !isDefined){ identifier\_list.extend = [] , identifier\_list.extend.append(item) } else{cout<<ID<<””at line”<<ID.lineno<<"has already been defined" ; identifier\_list.type = "type\_error" }}| ID{ identifier\_list.str = identifier\_list1.str ; item["name"] = identifier\_list1 , item["type"] = "" , item["F\_type"] = False , item["F\_const"] = False , item["declare\_line"] = identifier\_list1.lineno , item["extend"] = [] , isDefined = symboltable.find(identifier\_list1, mode=1) ; if(!isRecord && !isDefined){ symboltable.insert(item) } else if(isRecord && !isDefined){ identifier\_list.extend = [] , identifier\_list.extend.append(item) } else{ cout<< identifier\_list1<<””at line”<< identifier\_list1.lineno<<"has already been defined" ; identifier\_list.type = "type\_error" }}

const\_declarations -> CONST const\_declaration ; {cout<<””;”;if(const\_declaration.type== ””type\_error”){const\_declarations.type = ”type\_error”}}| empty

const\_declaration -> const\_declaration ; Q ID = const\_variable {cout<<””const”<< const\_variable.type << ID << “=” << const\_variable.str ; if(const\_variable.type != "type\_error" && const\_declaration.type != "type\_error"){if{ !symboltable.find(ID){ item["name"] = ID , item["type"] = const\_variable.type , item["F\_type"] = False , item["F\_const"] = True , item["declare\_line"] = ID.lineno , item["extend"] = const\_variable.value , symboltable.insert(item);} else{cout<<ID << "at line" << ID.lineno << "has already been defined" ; const\_declaration.type = "type\_error" ;} else{ const\_declaration.type = "type\_error" } } }}| ID = const\_variable{cout<< "const" << const\_variable.type << ID << “=” << const\_variable.str ; if(const\_variable.type != "type\_error"){ if( !symboltable.find(ID) ){ item["name"] = ID , item["type"] = const\_variable.type , item["F\_type"] = False , item["F\_const"] = True , item["declare\_line"] = ID.lineno , item["extend"] = const\_variable.value , symboltable.insert(item) ; } else{cout<< ID << "at line" << ID.lineno << "has already been defined" ; const\_declaration.type = “type\_error”;} }else{ const\_declaration.type = “type\_error” } }

Q -> empty;{cout<<”;”}

const\_variable -> + ID{ index = symboltable.find\_item(ID) ; if(!index){ const\_variable.type = "type\_error" ; cout<< ID << "at line" << ID.lineno << "is not a const variable" ; } else if(symboltable.table[index]["F\_const"] == False){cout<< "Bad const variable declared at line" << ID.lineno << ID << "is not a const variable" ; const\_variable.type = "type\_error"} else{ const\_variable.type = symboltable.table[index]["type"] ; const\_variable.str = “+” + ID ; const\_variable.val = symboltable.table[index]["extend"] ; const\_variable.lineno = ID.lineno ; } }

const\_variable -> - ID{ index = symboltable.find\_item(ID) ; if(!index){ const\_variable.type = "type\_error" ; cout<< ID << "at line" << ID.lineno << "is not a const variable" ; } else if(symboltable.table[index]["F\_const"] == False){cout<< "Bad const variable declared at line" << ID.lineno << ID << "is not a const variable" ; const\_variable.type = "type\_error"} else{ const\_variable.type = symboltable.table[index]["type"] ; const\_variable.str = “-” + ID ; const\_variable.val = symboltable.table[index]["extend"] ; const\_variable.lineno = ID.lineno ; } }

const\_variable -> + NUM{const\_variable.value = NUM ; const\_variable.str = “+” + NUM.str ; const\_variable.type = "int" ; const\_variable.lineno = NUM.lineno ; }

const\_variable -> - NUM{const\_variable.value = -NUM ; const\_variable.str = “-” + NUM.str ; const\_variable.type = "int" ; const\_variable.lineno = NUM.lineno ; }

const\_variable -> NUM{const\_variable.value = NUM ; const\_variable.str = NUM.str ; const\_variable.type = "int" ; const\_variable.lineno = NUM.lineno ; }

const\_variable -> CHARACTER{const\_variable.value = CHARACTER; const\_variable.str = CHARACTER.str ; const\_variable.type = "char" ; const\_variable.lineno = CHARACTER.lineno ; }

const\_variable -> + FLOAT{const\_variable.value = FLOAT; const\_variable.str = “+” + FLOAT.str ; const\_variable.type = "float" ; const\_variable.lineno = FLOAT.lineno ; }

const\_variable -> - FLOAT{const\_variable.value = -FLOAT; const\_variable.str = “-” + FLOAT.str ; const\_variable.type = "float" ; const\_variable.lineno = FLOAT.lineno ; }

const\_variable -> FLOAT{const\_variable.value = FLOAT; const\_variable.str = FLOAT.str ; const\_variable.type = "float" ; const\_variable.lineno = FLOAT.lineno ; }

const\_variable -> ID{index = symboltable.find\_item(ID) ; if(index == None){cout<< ID << "at line" << ID.lineno << "is not a const variable" ; const\_variable.type = "type\_error" ; } else{if(symboltable.table[index]["F\_const"] == False){cout<< ID << "at line" << ID.lineno << "is not a const variable" ;} else{const\_variable.type = symboltable.table[index]["type"] ; const\_variable.str = ID ; const\_variable.value = symboltable.table[index]["extend"] ; }} const\_variable.lineno = ID. lineno ; }

type\_declarations -> TYPE R type\_declaration {if(type\_declaration.type = "type\_error"){ type\_declarations.type = "type\_error" ; } }; empty

R -> empty

type\_declaration -> type\_declaration ; S ID = type {if(type.type[0:5] != “array”){cout<< "typedef" << type.type << ID << “;” ; } else{cout<< "typedef" << type.type[5:] << ID ; } for(i = 1; i <= len(type.extend); i++){cout<< “[” << type. extend[i][1] << “];”} if(type.type != "type\_error"){if(!symboltable.find(ID){ item["name"] = ID , item["type"] = type.type , item["F\_type"] = True , item["F\_const"] = False , item["declare\_line"] = S.lineno , item["extend"] = type.extend , symboltable.insert(item) ; } else{cout<< ID << "at line" << ID.lineno << "has been declared" ; type\_declaration.type = “type\_error” ; } else{ p[0].type = "type\_error" ; } }| ID = type{if(type.type[0:5] != “array”){cout<< "typedef" << type.type << ID << “;” ; } else{cout<< "typedef" << type.type[5:] << ID ; } for(i = 1; i <= len(type.extend); i++){cout<< “[” << type. extend[i][1] << “];”} if(type.type != "type\_error"){if(!symboltable.find(ID){ item["name"] = ID , item["type"] = type.type , item["F\_type"] = True , item["F\_const"] = False , item["declare\_line"] = S.lineno , item["extend"] = type.extend , symboltable.insert(item) ; } else{cout<< ID << "at line" << ID.lineno << "has been declared" ; type\_declaration.type = “type\_error” ; } else{ p[0].type = "type\_error" ; } }

S -> empty

type -> standard\_type {type.type = standard\_type. Type ; type.str = standard\_type.str ; }| RECORD L record\_body END {type.str = "struct\n{\n" + record\_body.str + "\n}" ; if(record\_body.type != "type\_error"){type.type = "record" ; type.extend = record\_body.extend ; }else{type.type = "type\_error" ; } }| ARRAY[periods] OF type{if(periods.type != "type\_error" && type1.type != "type\_error"){type.type = "array" + type1.type ; type.str = type1.str ; type.extend = [type1.type] + periods.extend ; } else{type.type = "type\_error" ;} }

type -> ID{ index = symboltable.find\_item(ID) ; if(!index){cout<< ID << "at line" << ID.lineno << "is not a type" ; type.type = "type\_error" ;} else{ flag1 = symboltable.find(ID) ; flag2 = symboltable.table[index]["F\_type"] ; if(flag1 && flag2){ type.name = ID , type.str = ID , type.type = ID , type.extend = "" ;} else{type.type = "type\_error" ; if(!flag2){cout<< "Bad type declaration at line" << ID.lineno << ID << "is not a type" ; } if(!flag1){cout<< ID << "at line" << ID.lineno << "is not declared" ; } } } }

L -> empty{L.extend = [] ; cout<< “,” ; }

standard\_type -> INTEGER{standard\_type.str = “int” ; standard\_type.type = “int” ; }

standard\_type -> REAL{standard\_type.str = “float” ; standard\_type.type = “float” ; }

standard\_type -> BOOLEAN{standard\_type.str = “int” ; standard\_type.type = “int” ; }

standard\_type -> CHAR{standard\_type.str = “char” ; standard\_type.type = “char” ; }

record\_body -> var\_declaration{record\_body.str = var\_declaration.str ; if(var\_declaration.type != "type\_error"){record\_body.extend = var\_declaration.extend ; } else{record\_body.type = "type\_error" ;} }

record\_body -> empty{record\_body.str = “” ; record\_body.extend = [] ; }

periods -> periods , period {if(periods1.type != "type\_error" && period.type != "type\_error"){periods.extend = periods1.extend + period.extend ;} else{periods.type = "type\_error" ; } }| period{if(periods1.type != "type\_error"){periods.extend = periods1.extend ; } else{periods.type = "type\_error" ; } }

period -> const\_variable TOTO const\_variable{if(const\_variable1.type == “int” && const\_variable2.type == “int” && const\_variable1.value < const\_variable2.value){period.extend = [((const\_variable1.value, const\_variable2.value - const\_variable1.value + 1))] ; } else{cout<< "Bad array index declaration at line" << TOTO.lineno ; period.type = "type\_error" ;} }

var\_declarations -> VAR var\_declaration {cout<< var\_declaration.str ; if(var\_declaration.type == “type\_error”){var\_declarations.type = "type\_error" ; } }| empty

var\_declaration -> var\_declaration identifier\_list : type ; {if(!isRecord){if(type.type == “type\_error” || var\_declaration1.type == "type\_error"){var\_declaration.type = "type\_error" ; } else if(type.type[0:5] == “array”){idx = symboltable.Top - 1 ; for(i = 0; i <= id\_cnt; i++){ symboltable.table[idx - i]["type"] = type.type ; symboltable.table[idx - i]["extend"] = type.extend ; } var\_declaration.str = var\_declaration1.str + "\n" + type.extend[0] ; temp = identifier\_list.str.split(',', 1) ; for(str in temp){Temp = "" ; for(i = 1; i <= len(type.extend); i++){Temp = Temp + "[" + str(type.extend[i][1]) + "]" ; } var\_declaration.str = var\_declaration.str + Str + Temp + “,” ; } var\_declaration.str = var\_declaration.str[:-1] + “;” ;} else{ idx = symboltable.Top - 1 ; for(i = 0; i <= id\_cnt; i++){ symboltable.table[idx-i]["type"] = type.type , symboltable.table[idx-i]["extend"] = type.extend ; } var\_declaration.str = var\_declaration1.str + "\n" + type.str + " " + identifier\_list.str + ";" ; } } else{ list = identifier\_list.extend ; length = len(list) ; for(i = length - id\_cnt; i <= length; i++){ list[i]["type"] = type.type , list[i]["extend"] = type.extend ;} var\_declaration.extend = var\_declaration1.extend + list ; if(type.type == "type\_error" || var\_declaration1.type == "type\_error"){ var\_declaration.type = "type\_error" ;} else if(type.type[0:5] == "array"){ var\_declaration.str = var\_declaration1.str + "\n" + type.extend[0] + " " ; temp = identifier\_list.str.split(',',1) ; for(Str in temp){ Temp = "" ; for(i = 1; i <= len(p[4].extend; i++){ Temp = Temp + "[" + str(type.extend[i][1])+"]" ; } var\_declaration.str = var\_declaration.str + Str + Temp + "," ;} var\_declaration.str = var\_declaration.str[:-1] + ";" ; } else{ var\_declaration.str = var\_declaration1.str + "\n" + type.str + " " + identifier\_list.str + ";" ; } } id\_cnt = 0 ; }| identifier\_list : type ;{if(!isRecord){if(type.type == "type\_error" || identifier\_list.type == "type\_error"){ var\_declaration.type = "type\_error" ; } else if(type.type[0:5] == "array"){ idx = symboltable.Top – 1 ; for(i = 0; i <= id\_cnt; i++){ symboltable.table[idx - i]["type"] = type.type , symboltable.table[idx - i]["extend"] = type.extend} ; } var\_declaration.str = type.extend[0] + " " ; temp = identifier\_list.str.split(',',1) ; for(Str in temp){ Temp = "" ; for(i = 1; i <= len(type.extend; i++){ Temp = Temp + "[" + str(type.extend[i][1]) + "]" ; } var\_declaration.str = var\_declaration.str + Str + Temp + "," ; } var\_declaration.str = var\_declaration.str[:-1] + ";" ; } else{ idx = symboltable.Top - 1 ; for(i = 0; i <= id\_cnt; i++){ symboltable.table[idx - i]["type"] = type.type , symboltable.table[idx - i]["extend"] = type.extend ;} var\_declaration.str = type.str + " " + identifier\_list.str + ";" ; } else{ list = identifier\_list.extend ; length = len(list) ; for(i = length - id\_cnt; i <= length; i++){ list[i]["type"] = type.type ; list[i]["extend"] = type.extend ; } var\_declaration.extend = list ; if(type.type == "type\_error" || identifier\_list.type == "type\_error"){ var\_declaration.type = "type\_error" ; } else if(type.type[0:5] == "array"){ var\_declaration.str = type.extend[0] + " " ; temp = identifier\_list.str.split(',',1) ; for(Str in temp){ Temp = "" ; for(i = 1; i <= len(p[3].extend); i++){ Temp = Temp + "[" + str(type.extend[i][1]) + "]" ; } var\_declaration.str = var\_declaration.str + Str + Temp + "," ; } var\_declaration.str = var\_declaration.str[:-1] + ";" ; } else{ var\_declaration.str = type.str + " " + identifier\_list.str + ";" ; } } id\_cnt = 0 ;}

subprogram\_declarations -> subprogram\_declarations subprogram\_declaration ; {if(subprogram\_declarations1.type == "type\_error" || subprogram\_declaration.type == "type\_error"){ subprogram\_declarations.type = "type\_error" ; } }| empty

subprogram\_declaration -> subprogram\_head subprogram\_body{if(subprogram\_head.type == "type\_error" or subprogram\_body.type == "type\_error"){ subprogram\_declaration.type = "type\_error" ; } }

subprogram\_body -> T const\_declarations type\_declarations var\_declarations compound\_statement LC\_U{if(subprogram\_head.type != "void"){cout<< "return" << subprogram\_head.name << “;” ;} cout<< “}” ; if(const\_declarations.type == "type\_error" or type\_declarations.type == "type\_error" or var\_declarations.type == "type\_error" or compound\_statement.type == "type\_error"){ subprogram\_body.type = "type\_error" ; } }

T -> empty{cout<< “{” ; if(subprogram\_head.type != “void” ){cout<< subprogram\_head.type << subprogram\_head.name << ";" ; } }

LC\_U -> empty{cout<< compound\_statement.str ;}

subprogram\_head -> X FUNCTION ID formal\_parameter : standard\_type ;{ hasDefined = symboltable.find(ID, mode=1) ; if(!hasDefined && formal\_parameter.type != "type\_error"){ fun\_name = ID + "\_function" ; cout<< standard\_type.str << fun\_name << formal\_parameter.str ;} subprogram\_head.name = ID ; subprogram\_head.type = standard\_type.type ; item = symboltable.domain\_stack[1] ; symboltable.table[item]["name"] = ID ; symboltable.table[item]["type"] = "function" ; symboltable.table[item]["F\_type"] = False ; symboltable.table[item]["F\_const"] = False ; symboltable.table[item]["declare\_line"] = ID.lineno ; symboltable.table[item]["extend"] = [standard\_type.type] + formal\_parameter.parameter\_list ; else if(hasDefined){cout<< ID << "at line" << ID.lineno << "has already been defined" ; subprogram\_head.type = "type\_error" ; } else{ subprogram\_head.type = "type\_error" ; } }

X -> empty{ item["name"] = "" ; item["type"] = "function" ; item["F\_type"] = False ; item["F\_const"] = False ; item["declare\_line"] = 0 ; symboltable.locate() ; symboltable.insert(item) ; }

subprogram\_head -> X FUNCTION ID formal\_parameter : array\_type ;{ hasDefined = symboltable.find(ID, mode=1) ; if(!hasDefined && formal\_parameter.type != "type\_error" && array\_type.type != "type\_error"){ fun\_name = ID + "\_function" ; cout<< array\_type.str[5:] << “\*” << fun\_name << formal\_parameter.str ; subprogram\_head.name = ID ; subprogram\_head.type = array\_type.str ; item = symboltable.domain\_stack[1] ; symboltable.table[item]["name"] = ID ; symboltable.table[item]["type"] = "function" ; symboltable.table[item]["F\_type"] = False ; symboltable.table[item]["F\_const"] = False ; symboltable.table[item]["declare\_line"] = ID.lineno ; symboltable.table[item]["extend"] = [array\_type.parameter\_list] + formal\_parameter.parameter\_list ; symboltable.locate() ; symboltable.insert(item) ; } else if(hasDefined){cout<< FUNCTION << "at line" << FUNCTION.lineno << "has already been defined" ; subprogram\_head.type = "type\_error" ; } else{ subprogram\_head.type = "type\_error"} }

subprogram\_head -> X FUNCTION ID formal\_parameter : ID ;{ index = symboltable.find\_item(ID1) ; if(!index){cout<< ID1 << "at line" << ID1.lineno << "is not a type" ; subprogram\_head.type = "type\_error" ;} else{if(symboltable.table[index]["F\_type"] == False){cout<< "Return type error at line" << ID1.lineno << ID1 << "is not a type" ; subprogram\_head.type = "type\_error" ; } else{ hasDefined = symboltable.find(ID, mode=1) ; if(!hasDefined && formal\_parameter.type == "type\_error"){ fun\_name = ID + "\_function"; cout<< ID1 << fun\_name << formal\_parameter.str ; subprogram\_head.name = ID ; subprogram\_head.type = ID1 ; item = symboltable.domain\_stack[1] ; symboltable.table[item]["name"] = ID ; symboltable.table[item]["type"] = "function" ; symboltable.table[item]["F\_type"] = False ; symboltable.table[item]["F\_const"] = False ; symboltable.table[item]["declare\_line"] = ID.lineno ; symboltable.table[item]["extend"] = [ID1] + formal\_parameter.parameter\_list ; } else if(hasDefined){cout<< ID << "at line" << ID.lineno << "has already been defined" ; subprogram\_head.type = "type\_error" ; } else{ subprogram\_head.type = "type\_error" ; } } } }

subprogram\_head -> X PROCEDURE ID formal\_parameter ;{ hasDefined = symboltable.find(ID, mode=1) ; if(!hasDefined && formal\_parameter.type != "type\_error"){ fun\_name = ID + "\_function" ; cout<< "void" << fun\_name << formal\_parameter.str ; subprogram\_head.name = ID ; subprogram\_head.type = "void" ; item = symboltable.domain\_stack[1] ; symboltable.table[item]["name"] = ID ; symboltable.table[item]["type"] = "procedure" ; symboltable.table[item]["F\_type"] = False ; symboltable.table[item]["F\_const"] = False ; symboltable.table[item]["declare\_line"] = ID.lineno ; symboltable.table[item]["extend"] = formal\_parameter.parameter\_list ; } else if(hasDefined){cout<< ID << "at line" << ID.lineno << "has already been defined" ; subprogram\_head.type = "type\_error" ; } else{ subprogram\_head.type = "type\_error"} }

formal\_parameter -> ( parameter\_lists ) { formal\_parameter.str = "(" + parameter\_lists.str + ")" ; formal\_parameter.parameter\_list = parameter\_lists.parameter\_list ; if(parameter\_lists.type == "type\_error"){ formal\_parameter.type = "type\_error" } else{ formal\_parameter.str = "()" ; formal\_parameter.parameter\_list = []} }| empty

parameter\_lists -> parameter\_lists ; parameter\_list { parameter\_lists.str = parameter\_lists1.str + "," + parameter\_list.str ; parameter\_lists.parameter\_list = parameter\_lists1.parameter\_list + parameter\_list.parameter\_list ; if(parameter\_lists1. type == "type\_error" && parameter\_list.type == "type\_error"){ parameter\_lists.type = "type\_error" ; } }| parameter\_list{ parameter\_lists.str = parameter\_lists1.str ; parameter\_lists.parameter\_list = parameter\_lists1.parameter\_list ; parameter\_lists.type = parameter\_lists1.type ;}

parameter\_list -> var\_parameter{parameter\_list.str = var\_parameter.str ; parameter\_list.parameter\_list = var\_parameter.parameter\_list ; if(var\_parameter.type == "type\_error"){parameter\_list.type = "type\_error" ; } }

parameter\_list -> value\_parameter{temp = value\_parameter.str.split(',') ; parameter\_list.str = "" ; for(Str in temp){if("array" in value\_parameter.type){ parameter\_list.str += value\_parameter.type[5:] + " " + Str + "[]," ; } else{parameter\_list.str += value\_parameter.type + " " + Str + "," ; } } parameter\_list.str = parameter\_list.str[:-1] ; parameter\_list.parameter\_list = value\_parameter.parameter\_list ; if(value\_parameter.type == "type\_error"){parameter\_list.type = "type\_error" ; } }

var\_parameter -> VAR value\_parameter{temp = value\_parameter.split(',',1) ; for(Str in temp){ var\_parameter.str = value\_parameter.type + "\* " + Str + "," ; } var\_parameter.str = var\_parameter.str[:-1] ; var\_parameter.parameter\_list = value\_parameter.parameter\_list ; if(value\_parameter.type == "type\_error"){var\_parameter.type = "type\_error" ; } }

value\_parameter -> identifier\_list : standard\_type{if(identifier\_list.type != "type\_error"){value\_parameter.str = identifier\_list.str ; value\_parameter.type = standard\_type.type ; idx = symboltable.Top ; for(i = idx - id\_cnt; i <= idx; i++){symboltable.table[idx]["type"] = standard\_type.type ; } i = id\_cnt ; value\_parameter.parameter\_list = [] ; while(i != 0){ value\_parameter.parameter\_list.append(standard\_type.type) ; i -= 1 ; } id\_cnt = 0 ; } else{value\_parameter.type = "type\_error" ; } }

value\_parameter -> identifier\_list : array\_type{if(identifier\_list.type != "type\_error" and array\_type.type != "type\_error"){value\_parameter.str = identifier\_list.str ; value\_parameter.type = "array" + array\_type.type ; i = id\_cnt ; value\_parameter.parameter\_list = [] ; while(i != 0){ value\_parameter.parameter\_list.append(array\_type.parameter\_list) ; i -= 1 ; } for(i = symboltable.Top-id\_cnt; i <= symboltable.Top; i++){symboltable.table[index]["type"] = value\_parameter.type ; symboltable.table[index]["extend"] = [] ; symboltable.table[index]["extend"].append(array\_type.type) ; symboltable.table[index]["extend"].append(None) ; } id\_cnt = 0 ; } else{value\_parameter.type = "type\_error" ; } }

value\_parameter -> identifier\_list : ID{index = symboltable.find\_item(ID) ; if(!index){cout<< ID << "at line" << ID.lineno << "is not a type" ; value\_parameter.type = "type\_error" ; id\_cnt = 0 ; } else{isType = symboltable.table[index]["F\_type"] ; if(isType && identifier\_list.type != "type\_error"){value\_parameter.str = identifier\_list.str ; value\_parameter.type = ID ; i = id\_cnt ; value\_parameter.parameter\_list = [] ; while(i != 0){ value\_parameter.parameter\_list.append(ID) ; i -= 1 ; } idx = symboltable.Top ; for(i = idx-id\_cnt; i <= idx; i++){symboltable.table[i]["type"] = ID ; } id\_cnt = 0 ; } else if(!isType){cout<< ID << "at line" << ID.lineno << "is not a type" ; value\_parameter.type = "type\_error" ; } else{value\_parameter.type = "type\_error" ; } } }

array\_type -> ARRAY OF standard\_type{array\_type.str = "array" + standard\_type.str ; array\_type.type = standard\_type.type ; array\_type.parameter\_list = "array" + standard\_type.type ; }

compound\_statement -> BEGIN statement\_list END U{if(statement\_list.type == "type\_error"){compound\_statement.type = "type\_error" ; } compound\_statement.str = statement\_list.str ; }

U -> empty

statement\_list -> statement\_list ; statement {statement\_list.str = statement\_list.str + statement.str ; if(statement\_list.type == "type\_error" || statement.type == "type\_error"){statement\_list.type = "type\_error"} }| statement{statement\_list.str = statement.str ; if(statement.type == "type\_error"){statement\_list.type = "type\_error" ; } }

statement -> variable ASSIGNOP expression{statement.str = variable.str + "=" + expression.str + ";\n" ; if(variable.type != "function"){if(variable.type != expression.type){if(variable.type != "type\_error" and expression.type != "type\_error"){cout<< variable.name << "at line" << variable.lineno << "type does not match" ; } statement.type = "type\_error" ; } } else{list = symboltable.get(variable.name) ; if(!list){cout<< variable.name << "at line" << variable.lineno << "does not have an item" ; statement.type = "type\_error" ; } else{if(list["extend"][0] != expression.type){if(expression.type != "type\_error"){cout<< variable.name << "at line" << variable.lineno << "type does not match" ; } statement.type = "type\_error" ; } } } }

statement -> call\_procedure\_statement{statement.str = call\_procedure\_statement.str ; statement.type = call\_procedure\_statement.type ; }

statement -> compound\_statement{statement.str = compound\_statement.str ; statement.type = compound\_statement.type ; }

statement -> IF expression THEN statement else\_part{statement.str = "if(" + expression.str + ")\n{\n" + statement1.str + "}" + else\_part.str ; if(expression.type != "boolean"){cout<< expression.str << "at line" << expression.lineno << "is not a boolean variable" ; statement.type = "type\_error" ; } else if(statement1.type == "type\_error" || else\_part.type == "type\_error"){statement.type = "type\_error" ; } }

statement -> CASE expression OF case\_body END{statement.str = "\nswitch(" + expression.str + ")\n{\n" + case\_body.str + "}\n" ; if(expression.type == "type\_error" or case\_body.type == "type\_error"){statement.type = "type\_error" ; } else{statement.type = case\_body.type ; } }

statement -> WHILE expression DO statement{statement.str = "while(" + expression.str + ")\n{\n" + statement1.str + '}\n' ; if(expression.type != "boolean"){cout<< expression.str << "at line" << expression.lineno << "is not a boolean variable" ; statement.type = "type\_error" ; } else{statement.type = statement1.type ; } }

statement -> REPEAT statement\_list UNTIL expression{statement.str = "do{\n" + statement\_list.str + "}\nwhile(" + expression.str + ")\n" ; if(expression.type != "boolean"){cout<< expression.str << "at line" << expression.lineno << "is not a boolean type" ; statement.type = "type\_error" ; } else{statement.type = statement\_list.type ; } }

statement -> FOR ID ASSIGNOP expression updown expression DO statement{if(updown.str == "to"){statement.str = "for(" + ID + "=" + expression.str + ";" + ID + "<=" + expression1.str + ";" + ID + "++)\n{\n" + statement1.str + "}\n" ; } else if(updown.str == "downto"){statement.str = "for(" + ID + "=" + expression.str + ";" + ID + ">=" + expression1.str + ";" + ID + "--)\n{\n" + statement1.str + "}\n" ; } if(!symboltable.find(ID)){cout<< ID << "is not declared at line" << ID.lineno ; statement.type = "type\_error" ; } if(symboltable.get\_type(ID) != expression.type && expression1.type != expression.type){cout<< "expressions'types conflict at line" << FOR.lineno ; statement.type = "type\_error" ; } else{statement.type = statement1.type ; } }

statement -> empty{statement.str = "" ; }

variable -> ID id\_varparts{variable.lineno = ID.lineno ; if(!symboltable.find(ID)){cout<< ID << "not defined at line" << ID.lineno ; variable.type = "type\_error" ; } else{variable.str = ID + id\_varparts.str ; variable.type = id\_varparts.type ; variable.name = ID ; variable.parameter\_list = [] ; variable.parameter\_list.append((variable.str, variable.type)) ; } }

id\_varparts -> id\_varparts id\_varpart {idtype = symboltable.get\_real\_type(ID) ; if(idtype == None){cout<< ID << "at line" << id\_varpart.lineno << "does not have an additional part" ; id\_varparts.type = "type\_error" ; } else if(idtype[0:5] != "array" && idtype != "record"){cout<< ID << "at line" << id\_varpart.lineno << "does not have an additional part" ; id\_varparts.type = "type\_error" ; } else{ id\_varparts.sublist = id\_varpart.sublist ; id\_varparts.str = id\_varparts1.str + id\_varpart.str ; if(id\_varparts1.type == "type\_error" || id\_varpart.type == "type\_error"){ id\_varparts.type = "type\_error" ; } else{ id\_varparts.type = id\_varpart.type ; } } }| empty{ id\_varparts.str = "" ; if(symboltable.find(ID)){ list = symboltable.get(ID) ; if(!list){ id\_varparts.type = "type\_error" ; } else{ id\_varparts.type = list["type"] ; id\_varparts.sublist = symboltable.get\_extend(ID) ; if(id\_varparts.sublist == None){ id\_varparts.sublist = [] ; } } } else{ id\_varparts.type = "type\_error" ; } }

id\_varpart -> [ expression\_list ] { id\_varpart.lineno = [.lineno ; id\_varpart.str = "[" + expression\_list.str + "]";if(len(id\_varparts.sublist) == 0){ id\_varpart.type = "type\_error" ; } else if(expression\_list.type == "int"){ id\_varpart.type = id\_varparts.sublist[0] ; } else{cout<< "array index type error at line" << [.lineno ; id\_varpart.type = "type\_error" ; } }| . ID{ id\_varpart.str = "." + ID + ";" ; id\_varpart.lineno = ID.lineno ; sublist = find\_type(id\_varparts.sublist, ID) ; if(!sublist){cout<< "record part not defined at line" << ID.lineno ; id\_varpart.type = "type\_error" ; } else{ id\_varpart.type = sublist["type"] ; id\_varpart.sublist = sublist["extend"] ; } }

else\_part -> ELSE statement { else\_part.str = "else\n{" + statement.str + "\n}" ; else\_part.type = statement.type ; }| empty{ else\_part.str = "" ;}

case\_body -> branch\_list ; { case\_body.str = branch\_list.str ; case\_body.type = branch\_list.type ; }| empty{ case\_body.str = "" ; case\_body.type = "" ; }

branch\_list -> branch\_list ; LZ branch { branch\_list.str = branch\_list1.str + branch.str ; if(branch\_list1.type == "type\_error" || branch.type == "type\_error"){ branch\_list.type = "type\_error" ; } }| LX branch{ branch\_list.str = branch.str ; branch\_list.type = branch.type ; }

LX -> empty{ LX.type = expression.type ; }

LZ -> empty{ LZ.type = expression.type ;}

branch -> const\_list : statement{ branch.str = "case " + const\_list.str + ":\n" + statement.str + "break;\n" ; if(const\_list.type != "type\_error" && const\_list.type == LZ.type){ branch.type = statement.type ; } else{cout<< "types conflict in casebody, at line" << const\_list.lineno ; branch.type = "type\_error" ; } }

const\_list -> const\_list , const\_variable { const\_list.str = const\_list.str + "," + const\_variable.str ; const\_list.lineno = ,.lineno ; if(const\_list.type == "type\_error"){ const\_list.type = "type\_error" ; } else if(const\_list.type != const\_variable.type){cout<< "types conflict at line" << const\_variable.lineno ; const\_list.type = "type\_error" ; } else{ const\_list.type = const\_list.type ; } else{ const\_list.type = const\_list.type ; } }| const\_variable{ const\_list.str = const\_variable.str ; const\_list.type = const\_variable.type ; const\_list.lineno = const\_variable.lineno ; }

updown -> TO{ updown.str = "to" ; updown.lineno = TO.lineno ; }

updown -> DOWNTO{ updown.str = "downto" ; updown.lineno = DOWNTO.lineno ; }

call\_procedure\_statement -> ID{call\_procedure\_statement.str = ID + "\_function();\n" ; if(!symboltable.find(ID)){cout<< ID << "at line" << ID.lineno << "is not defined" ; call\_procedure\_statement.type = "type\_error" ; } else if(symboltable.get\_type(ID) != "procedure"){cout<< ID << "at line" << ID.lineno << "is not a procedure" ; call\_procedure\_statement.type = "type\_error" ; } }

call\_procedure\_statement -> ID(expression\_list){ call\_procedure\_statement.str = ID + "\_function(" + expression\_list.str + ");\n" ; if(!symboltable.find(ID)){cout<< "function" << ID << "not defined at line" << ID.lineno ; call\_procedure\_statement.type = "type\_error" ; } else if(symboltable.get\_type(ID) == "function"){if(expression\_list.type != "".join(symboltable.get(ID)['extend'][1:])){cout<< "The parameters do not match at line" << ID.lineno ; call\_procedure\_statement.type = "type\_error" ; } } else if(symboltable.get\_type(ID) == "procedure"){if(expression\_list.type != "".join(symboltable.get(ID)['extend'])){cout<< "The parameters do not match at line" << ID.lineno ; call\_procedure\_statement.type = "type\_error" ; } } else{cout<< ID << "at line" << ID.lineno << "is not a function or procedure" ; call\_procedure\_statement.type = "type\_error" ; } }

call\_procedure\_statement -> WRITE(expression\_list){ format\_str = "" ; id\_str = "" ; for(I = 0; i <= len(expression\_list.parameter\_list); i++){if(expression\_list.parameter\_list[i][1] == "int"){ format\_str += "%d" ; } else if(expression\_list.parameter\_list[i][1] == "boolean"){ format\_str += "%d" ; } else if(expression\_list.parameter\_list[i][1] == "char"){ format\_str += "%c" ; } else{ format\_str += "%f" ; } id\_str += "," + expression\_list.parameter\_list[i][0] ; } call\_procedure\_statement.str = "printf(\"" + format\_str+ "\"" + id\_str + ");\n" ; }

call\_procedure\_statement -> READ(variables){ format\_str = "" ; id\_str = "" ; for(i = 0; i <= len(variables.parameter\_list); i++){if(variables.parameter\_list[i][1] == "int"){ format\_str += "%d" ; } else if(variables.parameter\_list[i][1] == "boolean"){ format\_str += "%d" ; } else if(variables.parameter\_list[i][1] == "char"){ format\_str += "%c" ; } else{ format\_str += "%f" ; } id\_str += ",&" + variables.parameter\_list[i][0] ; } call\_procedure\_statement.str = "scanf(\"" + format\_str + "\"" + id\_str + ");\n" ; if(variables.type == "type\_error"){ call\_procedure\_statement.type = "type\_error" ; cout<< "read parameters type error at line" << READ.lineno ; } }

variables -> variables , variable { variables1.parameter\_list.append((variable.str, variable.type)) ; variables.parameter\_list = variables1.parameter\_list[:] ; if(variables1.type == "type\_error" || variable.type == "type\_error"){variables.type = "type\_error" ; } else{variables.parameter\_list = variables1.parameter\_list[:] ; variables.type = variables1.type ; } }| variable{variables.parameter\_list = variables1.parameter\_list[:] ; variables.type = variables1.type ; }

expression\_list -> expression\_list , expression {expression\_list = Attribute() ; expression\_list.str = expression\_list.str + ','+ expression.str ; expression\_list.type = str(expression\_list.type) + str(expression.type) ; expression\_list.parameter\_list = [] ; expression\_list.parameter\_list.append((expression.str, expression.type)) ; for(i in expression\_list.parameter\_list){ expression\_list.parameter\_list.append(i) ; } if(expression\_list.type == "type\_error" || expression.type == "type\_error"){expression\_list.type = "type\_error" ; } }| expression{expression\_list = Attribute() ; expression\_list.str = expression.str ; expression\_list.type = expression.type ; expression\_list.parameter\_list =[] ; expression\_list.parameter\_list.append((expression.str, expression.type)) ; }

expression -> simple\_expression RELOP simple\_expression {expression = Attribute() ; expression.str = simple\_expression.str + RELOP + simple\_expression1.str ; expression.lineno = RELOP.lineno ; if(simple\_expression.type == "type\_error" || simple\_expression1.type == "type\_error"){expression.type = "type\_error" ; } else if(simple\_expression.type != simple\_expression1.type){cout<< "types don't match at line" << RELOP.lineno ; expression.type = "type\_error" ; } else{expression.type = "boolean" ; } }| simple\_expression{ expression.str = simple\_expression.str ; expression.type = simple\_expression.type ; expression.lineno = simple\_expression.lineno ; }

expression -> simple\_expression = simple\_expression{expression.str = simple\_expression.str + "=" + simple\_expression1.str ; expression.lineno = =.lineno ; if(simple\_expression.type == "type\_error" || simple\_expression1.type == "type\_error"){expression.type = "type\_error" ; } else if(simple\_expression.type != simple\_expression1.type){cout<< "types don't match at line" << =.lineno ; expression.type == "type\_error" ; } expression.type = "boolean" ; }

simple\_expression -> term{ simple\_expression.str = term.str ; simple\_expression.type = term.type ; simple\_expression.lineno = term.lineno ; }

simple\_expression -> + term{ simple\_expression.str = “+” + term.str ; simple\_expression.type = term.type ; simple\_expression.lineno = +.lineno ; }

simple\_expression -> - term{ simple\_expression.str = “-” + term.str ; simple\_expression.type = term.type ; simple\_expression.lineno = -.lineno ; }

simple\_expression -> simple\_expression + term{simple\_expression.str = simple\_expression1.str + "+" + term.str ; simple\_expression.lineno = +.lineno ; if(simple\_expression1.type == term.type){ simple\_expression.type = term.type ; } else{cout<< "types conflict at line" << +.lineno ; simple\_expression.type = "type\_error" ; } }

simple\_expression -> simple\_expression - term{simple\_expression.str = simple\_expression1.str + "-" + term.str ; simple\_expression.lineno = -.lineno ; if(simple\_expression1.type == term.type){ simple\_expression.type = term.type ; } else{cout<< "types conflict at line" << +.lineno ; simple\_expression.type = "type\_error" ; } }

simple\_expression -> simple\_expression OR term{if(simple\_expression1.type == "boolean" && term.type == "boolean"){simple\_expression.str = simple\_expression1.str + "||" + term.str ; } else{simple\_expression.str = simple\_expression1.str + "|" + term.str ; } simple\_expression.lineno = OR.lineno ; if(simple\_expression1.type == term.type){if(simple\_expression1.type == "int"){simple\_expression.type = "int" ; } else if(simple\_expression1.type == "boolean"){simple\_expression.type = "boolean" ; } else{if(simple\_expression1.type != "type\_error" && term.type != "type\_error"){cout<< "Type error at line" << OR.lineno << ", the operands must be integer or boolean" ; } simple\_expression.type = "type\_error" ; } } else{if(simple\_expression1.type != "type\_error" and term.type != "type\_error"){cout<< "Types conflict at line" << OR.lineno << ", and the operands must be integer or boolean" ;} simple\_expression.type = "type\_error" ; } }

term -> term MULOP factor {term.lineno = MULOP.lineno ; if(MULOP == 'mod'){ STR = '%' ; if(term1.type == factor.type){if(term1.type == "int"){term.type = "int" ; } else{if(term1.type != "type\_error" && factor.type != "type\_error"){cout<< "Type error at line" << MULOP.lineno << ", the operands must be integer or boolean" ; } term.type = "type\_error" ; } } else if(MULOP == 'div'){ STR = '/' ; if(term1.type == factor.type){if(term1.type == "int"){term.type = "int" ; } else{if(term1.type != "type\_error" && factor.type != "type\_error"){cout<< "Type error at line" << MULOP.lineno << ", the operands must be integer or boolean" ; } term.type = "type\_error" ; } } else{if(term1.type != "type\_error" && factor.type != "type\_error"){cout<< "Types conflict at line" << MULOP.lineno << ", and the operands must be integer or boolean" ; } term.type = "type\_error" ; } } else if(MULOP == 'and'){ STR = '&' ; if(term1.type == factor.type){if(term1.type == "int"){term.type = "int" ; } else if(term1.type == "boolean"){STR = "&&" ; term.type = "boolean" ; } else{if(term1.type != "type\_error" && factor.type != "type\_error"){cout<< "Type error at line" << MULOP.lineno << ", the operands must be integer or boolean" ; } term.type = "type\_error" ; } } else{if(term1.type != "type\_error" and factor.type != "type\_error"){cout<< "Types conflict at line" << MULOP.lineno << ", and the operands must be integer or boolean" ; } term.type = "type\_error" ; } } else{ STR = MULOP ; } term.str = term1.str + STR + factor.str ; } }| factor{ term.str = factor.str ; term.type = factor.type ; term.lineno = factor.lineno ; }

factor -> unsign\_const\_variable{factor.str = unsign\_const\_variable.str ; factor.type = unsign\_const\_variable.type ; factor.lineno = unsign\_const\_variable.lineno ; }

factor -> variable{factor.str = variable.str ; factor.type = variable.type ; factor.lineno = variable.lineno ; factor.name = variable.name ; }

factor -> ID( expression\_list ){ factor.str = ID + "\_function(" + expression\_list.str + ")" ; factor.lineno = ID.lineno ; if(!symboltable.find(ID)){cout<< "function" << ID << "not defined at line", ID.lineno ; factor.type = "type\_error" ; } else if(symboltable.get\_type(ID) != "function"){cout<< ID << "at line" << ID.lineno << "is not a function" ; factor.type = "type\_error" ; } else if(expression\_list.type != "".join(symboltable.get(ID)['extend'][1:])){cout<< "function's parameters do not match at line" << ID.lineno ; factor.type = "type\_error" ; } else{factor.type = symboltable.get(ID)["extend"][0] ; } }

factor -> ( expression ){ factor.str = "(" + expression.str + ")" ; factor.type = expression.type ; factor.lineno = expression.lineno ; }

factor -> NOT factor{factor.str = "~" + factor1.str ; factor.type = factor1.type ; factor.lineno = NOT.lineno ; }

unsign\_const\_variable -> NUM{unsign\_const\_variable.value = NUM ; unsign\_const\_variable.str = str(NUM) ; unsign\_const\_variable.lineno = NUM.lineno ; unsign\_const\_variable.type = "int" ; }

unsign\_const\_variable -> CHARACTER{unsign\_const\_variable.value = CHARACTER ; unsign\_const\_variable.str = str(CHARACTER) ; unsign\_const\_variable.lineno = CHARACTER.lineno ; unsign\_const\_variable.type = "char" ; }

unsign\_const\_variable -> FLOAT{unsign\_const\_variable.value = FLOAT ; unsign\_const\_variable.str = str(FLOAT) ; unsign\_const\_variable.lineno = FLOAT.lineno ; unsign\_const\_variable.type = "float" ; }

//错误处理楠神说不用写

def p\_empty(p):

'''empty :'''

pass

def p\_error(p):

print("SyntaxError!", p)

program\_head -> PROGRAM ID error) ;

program\_head -> PROGRAM ID (error)

program\_head -> PROGRAM ID (identifier\_list error;)

program\_head -> PROGRAM ID (identifier\_list) error

identifier\_list -> error , ID

const\_declaration -> error ; Q ID = const\_variable

const\_declaration -> error = const\_variable

const\_declaration -> const\_declaration ; Q error = const\_variable

const\_declaration -> const\_declaration ; Q ID = error

const\_declaration -> ID = error

const\_declaration -> const\_declaration error Q ID = const\_variable

type\_declaration -> error ; S ID = type

type\_declaration -> error = type

type\_declaration -> type\_declaration ; S error = type

type\_declaration -> type\_declaration ; S ID error

type\_declaration -> ID = error

type\_declaration -> type\_declaration error S ID = type

type -> ARRAY error OF type

type -> ARRAY[periods] OF error

periods -> error , period

period -> error TOTO const\_variable

var\_declaration -> var\_declaration error : type ;

var\_declaration -> var\_declaration identifier\_list : error ;

var\_declaration -> error : type ;

var\_declaration -> identifier\_list : error ;

var\_declaration -> var\_declaration identifier\_list : type error

var\_declaration -> identifier\_list : type error

subprogram\_head -> X FUNCTION ID(error : standard\_type ;