Oberon-0 Analysis

BNF Grammar

```
module → module id ; decls stat_block end id .
stat_block → begin stat_seq
    | ϵ
decls → cnst_decl  type_decl  var_decl  proc_decl_seq
cnst_decl → const cnst_ass
    | ϵ
cnst_ass → id = expr ; cnst_ass
    | ϵ
type_decl → type type_ass
    | ϵ
type_ass → id = type ; type_ass
    | ϵ
type → id
    | type_arr
    | type_rec
    | integer
    | boolean
type_arr → array expr of type
type_rec → record field_list end
field_list → id : type ; field_list
    | ϵ
var_decl → var var_list
    | ϵ
var_list → id_list : type ; var_list
    | ϵ
proc_decl_seq → proc_decl ; proc_decl_seq
    | ϵ
proc_decl → proc_head ; proc_body
proc_head → procedure id proc_pars
proc_pars → ( par_list )
    | ϵ
par_list → par ext_par
par → var id_list : type
    | id_list : type
ext_par → ; par ext_par
    | ϵ
id_list → id ext_id
ext_id → , id ext_id
    | ϵ
proc_body → decls  stat_block  end id
stat_seq → stat ext_stat
ext_stat → ; stat ext_stat
    | ϵ
stat → id id_stat
    | if_stat
    | while_stat
id_stat → assignment
    | proc_call
assignment → sel := expr
sel → . id sel
    | [ expr ] sel
    | ϵ
proc_call → act_pars
act_pars → ( expr_list )
    | ϵ
expr_list → expr ext_expr
    | ϵ
ext_expr → , expr ext_expr
    | ϵ
if_stat → if expr then stat_seq elif_seq else_stat end
elif_seq → elsif expr then stat_seq elif_seq
    | ϵ
else_stat → else stat_seq
    | ϵ
while_stat → while expr do stat_seq end
expr → simple_expr post_expr
post_expr → rel_op simple_expr
    | ϵ
rel_op → = | # | <
    | <= | > | >=
simple_expr → + term post_term
    | - term post_term
    | term post_term
post_term → + term post_term
    | - term post_term
    | or term post_term
    | ϵ
term → factor post_factor
post_factor → * factor post_factor
    | & factor post_factor
    | div factor post_factor
    | mod factor post_factor
    | ϵ
factor → id sel
    | num_token
    | ( expr )
    | ∼ factor
```

Syntax Directed Translation

```
s → { env = null }
    module
    { for ( call in calls )   call.verify() }
    { module.g_mod.show() }
module → module id ;
    { decl = new Decl(id, new ModuleType()); env.put(decl) }
    { g_mod = new Module(id.lexeme); module.g_mod = g_mod }
    { cur = env; env = new Env(env); }
    { decls.g_mod = g_mod; }
    decls
    { g_proc = g_mod.add('Main'); stat_block.g_stats = g_proc :: add }
    stat_block end id .
    { env = cur }
stat_block → begin
    { stat_seq.g_stats = stat_block.g_stats }
    stat_seq
    | ϵ
decls → cnst_decl  type_decl  var_decl
    { proc_decl_seq.g_mod = decls.g_mod }
    proc_decl_seq
cnst_decl → const cnst_ass
    | ϵ
cnst_ass → id = expr
    { decl = new Decl(id, expr.type, { modifiers : [ Decl.Modifiers.Const ] }) }
    { env.put(id.lexeme, decl) }
    ; cnst_ass_1
    | ϵ
type_decl → type type_ass
    | ϵ
type_ass → id = type
    { decl = new Decl(id, new Typedef(type.val)) }
    { env.put(id.lexeme, decl) }
    ; type_ass_1
    | ϵ
type → id
    { decl = env.get(id.lexeme); type.val = decl.type.val }
    | type_arr
    { type.val = type_arr.val }
    | type_rec
    { type.val = type_rec.val }
    | integer
    { type.val = new IntegerType() }
    | boolean
    { type.val = new BooleanType() }
type_arr → array expr of type
    { type_arr.val = new ArrayType(type.val) }
type_rec → record
    { field_list.l_fields = { } }
    field_list end
    { type_rec.val = new RecType(field_list.fields) }
field_list → id : type ;
    { decl = new Decl(id, type.val, { modifiers : [ Decl.Modifiers.Var ] }) }
    { field_list.l_fields.put(id.lexeme, decl); field_list_1.l_fields = field_list.l_fields }
    field_list_1
    { field_list.fields = field_list_1.fields }
    | ϵ   { field_list.fields = field_list.l_fields }
var_decl → var var_list
    | ϵ
var_list → id_list : type ;
    {
      for(id in id_list.ids) {
        env.put(id.lexeme, new Decl(id, type.val, { modifiers : [ Decl.Modifieres.Var ] }));
      }
    }
    var_list_1
    | ϵ
proc_decl_seq → { proc_decl.g_mod = proc_decl_seq.g_mod }
    proc_decl ;
    { proc_decl_seq_1.g_mod = proc_decl_seq.g_mod }
    proc_decl_seq_1
    | ϵ
proc_decl → { cur = env; env = new Env(env) }
    proc_head ;
    { proc_body.g_mod = proc_decl.g_mod; proc_body.g_proc = proc_decl.g_mod.add(proc_head.id.lexeme) }
    { proc_body.decl_id = proc_head.id }
    proc_body
    { env = cur; env.put(proc_head.id, proc_head.decl) }
proc_head → procedure id proc_pars
    { decl = new Decl(id, new FunctionType(proc_pars.params)) }
    { proc_head.decl = decl; proc_head.id = id }
proc_pars → ( par_list )   { proc_pars.params = par_list.params }
    | ϵ   { proc_pars.params = [ ] }
par_list → par ext_par
    { par_list.params = [ ...par.params, ...ext_par.params ] }
par → { par.params = [ ] }
    var id_list : type
    {
      for(id in id_list.ids) {
        decl = new Decl(id, type.val, { modifiers : [ Decl.Modifiers.Var ] });
        par.params.push(decl);
        env.put(id.lexeme, decl);
      }
    }
    |{ par.params = [ ] }
    id_list : type
    {
      for(id in id_list.ids) {
        decl = new Decl(id, type.val);
        par.params.push(decl);
        env.put(id.lexeme, decl);
      }
    }
ext_par → ; par ext_par_1
    { ext_par.params = [ ...par.params, ...ext_par_1.params ] }
    | ϵ   { ext_par.params = [ ] }
id_list → id ext_id   { id_list.ids = [ id, ...ext_id.ids ] }
ext_id → , id ext_id_1   { ext_id.ids = [ id, ...ext_id_1.ids ] }
    | ϵ   { ext_id.ids = [ ] }
proc_body → { decls.g_mod = proc_body.g_mod }
    decls
    { stat_block.g_stats = proc_body.g_proc :: add }
    stat_block  end id
stat_seq → { stat.g_stats = stat_seq.g_stats }
    stat
    { ext_stat.g_stats = stat_seq.g_stats }
    ext_stat
ext_stat → ;   { stat.g_stats = ext_stat.g_stats }
    stat
    { ext_stat_1.g_stats = ext_stat.g_stats }
    ext_stat_1
    | ϵ
stat → id
    { id_stat.g_stats = stat.g_stats; id_stat.id = id }
    id_stat
    | if_stat   { stat.g_stats.add(if_stat.statement) }
    | while_stat   { stat.g_stats.add(while_stat.statement) }
id_stat → { assignment.decl = env.get(id_stat.id.getLexeme()); assignment.id = id_stat.id }
    assignment
    { id_stat.g_stats.add(assignment.statement) }
    | { proc_call.decl = env.get(id_stat.id.getLexeme()); proc_call.id = id_stat.id }
    proc_call
    { id_stat.g_stats.add(proc_call.statement) }
assignment → { decl = assignment.decl; sel.parent_type = decl.type }
    sel := expr
    { assignment.statement = new PrimitiveStatement(decl.lexeme || sel.text || ' :=' || expr.text) }
sel → . id
    { type = sel.parent_type.fields.get(id); sel_1.parent_type = type }
    sel_1
    { sel.text = '.' || id.lexeme || sel_1.text; sel.type = sel_1.type }
    | [ expr ]
    { sel_1.parent_type = sel.parent_type.type }
    sel_1
    { sel.text = '[' || expr.text || ']' || sel_1.text; sel.type = sel_1.type }
    | ϵ   { sel.text = ''; sel.type = sel.parent_type }
proc_call → { decl = proc_call.decl }
    act_pars
    { calls.add(new ProcCall(proc_call.id, act_pars.types)) }
    { proc_call.statement = new PrimitiveStatement(decl.lexeme || act_pars.text) }
act_pars → ( expr_list )
    { act_pars.types = expr_list.types }
    { act_pars.text = '(' || expr_list.text || ')' }
    | ϵ
    { act_pars.types = [ ] }
    { act_pars.text = '' }
expr_list → expr ext_expr
    { expr_list.types = [ expr.type, ...ext_expr.types ] }
    { expr_list.text = expr.text || ext_expr.text }
    | ϵ
    { expr_list.types = [ ] }
    { expr_list.text = '' }
ext_expr → , expr ext_expr
    { ext_expr_1.types = [ expr.type, ...ext_expr.types ] }
    { ext_expr.text = ',' || expr.text || ext_expr_1.text }
    | ϵ
    { ext_expr.types = [ ] }
    { ext_expr.text = '' }
if_stat → if expr
    { statement = new IfStatement(expr.text) }
    then
    { g_tbody = statement.getTrueBody(); stat_seq.g_stats = g_tbody :: add }
    stat_seq
    { g_fbody = statement.getFalseBody(); elif_seq.g_stats = g_fbody :: add }
    elif_seq
    { g_stats = elif_seq.g_pstats; else_stat.g_stats = g_stats }
    else_stat end
    { if_stat.statement = statement }
elif_seq → elsif expr
    { statement = new IfStatement(expr.text) }
    then
    { g_tbody = statement.getTrueBody(); stat_seq.g_stats = g_tbody :: add }
    stat_seq
    { g_fbody = statement.getFalseBody(); elif_seq_1.g_stats = g_fbody :: add }
    elif_seq_1
    { elif_seq.g_stats.add(statement); elif_seq.g_pstats = elif_seq_1.g_pstats }
    | ϵ   { elif_seq.g_pstats = elif_seq.g_stats }
else_stat → else
    { stat_seq.g_stats = else_stat.g_stats }
    stat_seq
    | ϵ
while_stat → while expr
    { statement = new WhileStatement(expr.text) }
    do
    { g_lbody = statement.getLoopBody(); stat_seq.g_stats = g_lbody :: add }
    stat_seq end
    { while_stat.statement = statement }
expr → simple_expr
    { post_expr.l_type = simple_expr.type }
    { post_expr.l_text = simple_expr.text }
    post_expr
    { expr.type = post_expr.type }
    { expr.text = post_expr.text }
post_expr → rel_op simple_expr
    { post_expr.text = post_expr.l_text || rel_op.text || simple_expr.text }
    { post_expr.type = new BooleanType() }
    | ϵ
    { post_expr.text = post_expr.l_text }
    { post_expr.type = post_expr.l_type }
rel_op → =   { rel_op.text = '=' }
    | #   { rel_op.text = '#' }
    | <   { rel_op.text = '<' }
    | <=   { rel_op.text = '<=' }
    | >   { rel_op.text = '>' }
    | >=   { rel_op.text = '>=' }
simple_expr → + term
    { post_term.l_type = new IntegerType() }
    { post_term.l_text = '(+' || term.text || ')' }
    post_term
    { simple_expr.type = post_term.type; simple_expr.text = post_term.text }
    | - term
    { post_term.l_type = new IntegerType() }
    { post_term.l_text = '(−' || term.text || ')' }
    post_term
    { simple_expr.type = post_term.type; simple_expr.text = post_term.text }
    | term
    { post_term.l_type = term.type; post_term.l_text = term.text }
    post_term
    { simple_expr.type = post_term.type; simple_expr.text = post_term.text }
post_term → + term
    { post_term_1.l_type = new IntegerType() }
    { post_term_1.l_text = post_term.l_text || '+' || term.text }
    post_term_1
    { post_term.type = post_term_1.type; post_term.text = post_term_1.text }
    | - term
    { post_term_1.l_type = new IntegerType() }
    { post_term_1.l_text = post_term.l_text || '−' || term.text }
    post_term_1
    { post_term.type = post_term_1.type; post_term.text = post_term_1.text }
    | or term
    { post_term_1.l_type = new BooleanType() }
    { post_term_1.l_text = post_term.l_text || 'or' || term.text }
    post_term_1
    { post_term.type = post_term_1.type; post_term.text = post_term_1.text }
    | ϵ
    { post_term.type = post_term.l_type }
    { post_term.text = post_term.l_text }
term → factor
    { post_factor.l_type = factor.type; post_factor.l_text = factor.text }
    post_factor
    { term.type = post_factor.type; term.text = post_factor.text }
post_factor → * factor
    { post_factor_1.l_type = new IntegerType() }
    { post_factor_1.l_text = post_factor.l_text || '*' || factor.text }
    post_factor_1
    { post_factor.type = post_factor_1.type; post_factor.text = post_factor_1.text }
    | & factor
    { post_factor_1.l_type = new BooleanType() }
    { post_factor_1.l_text = post_factor.l_text || '&' || factor.text }
    post_factor_1
    { post_factor.type = post_factor_1.type; post_factor.text = post_factor_1.text }
    | div factor
    { post_factor_1.l_type = new IntegerType() }
    { post_factor_1.l_text = post_factor.l_text || 'div' || factor.text }
    post_factor_1
    { post_factor.type = post_factor_1.type; post_factor.text = post_factor_1.text }
    | mod factor
    { post_factor_1.l_type = new IntegerType() }
    { post_factor_1.l_text = post_factor.l_text || 'mod' || factor.text }
    post_factor_1
    { post_factor.type = post_factor_1.type; post_factor.text = post_factor_1.text }
    | ϵ
    { post_factor.type = post_factor.l_type }
    { post_factor.text = post_factor.l_text }
factor → id
    { decl = env.get(id.lexeme); sel.parent_type = decl.type }
    sel
    { factor.type = sel.type; factor.text = id.lexeme || sel.text }
    | num_token
    { factor.type = new IntegerType(); factor.text = num_token.lexeme }
    | ( expr )
    { factor.type = expr.type; factor.text = '(' || expr.text || ')' }
    | ∼ factor_1
    { factor.type = new BooleanType(); factor.text = '∼' || factor_1.text }
```