

---

*Project:* DRA FRONT END FILTER PROJECT

*Title:* Specification of Query Transformations in SML (II)

*Ref:* DS/FMU/FEF/020      *Issue: Revision : 2.2*      *Date:* 5 December 2009

*Status:* Approved      *Type:* Specification

*Keywords:*

*Author:*

<i>Name</i>	<i>Location</i>	<i>Signature</i>	<i>Date</i>
G. M. Prout	WIN01		

*Authorisation for Issue:*

<i>Name</i>	<i>Function</i>	<i>Signature</i>	<i>Date</i>
R.B. Jones	HAT Manager		

*Abstract:* A specification of the SSQL Query Transformations in Standard ML for the DRA front end filter project RSRE 1C/6130.

*Distribution:* HAT FEF File  
Simon Wiseman

## 0 DOCUMENT CONTROL

### 0.1 Contents List

<b>0</b>	<b>DOCUMENT CONTROL</b>	<b>2</b>
0.1	Contents List . . . . .	2
0.2	Document Cross References . . . . .	2
0.3	Changes History . . . . .	2
0.4	Changes Forecast . . . . .	2
<b>1</b>	<b>GENERAL</b>	<b>3</b>
1.1	Scope . . . . .	3
1.2	Introduction . . . . .	3
<b>2</b>	<b>THE TRANSFORMATIONS</b>	<b>3</b>
2.1	Incompletely Specified Transformations . . . . .	3
2.2	Symbol Table Model . . . . .	5
2.3	Symbol Table Operations . . . . .	15
2.4	Transformations Proper . . . . .	19
<b>3</b>	<b>INDEX</b>	<b>82</b>

### 0.2 Document Cross References

- [1] L.Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
- [2] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [3] DS/FMU/FEF/018. *Proposal for Phase 2*. G.M. Prout, ICL Secure Systems, WIN01.
- [4] DS/FMU/FEF/019. *Specification of Query Transformations in SML (I)*. G.M. Prout, ICL Secure Systems, WIN01.
- [5] *SSQL Transformations*. Simon Wiseman, DRA, 14th January 1993.

### 0.3 Changes History

**Issue 1.1 (28 January 1993)** First draft.

**Issue 1.2 (29 January 1993)** Mutual recursion dealt with by specifying appropriate functions together.

**Issue Revision : 2.2 (5 December 2009)** Final approved version.

**Issue 2.2** Removed dependency on ICL logo font

### 0.4 Changes Forecast

None.

## 1 GENERAL

### 1.1 Scope

This document gives a formal specification in Standard ML ([2], [1]) of the SSQL query transformations of [5]. It constitutes part of deliverable D8 of work package 3, as given in the Proposal for Phase 2, [3].

### 1.2 Introduction

We provide Standard ML specifications of the SSQL query transformations of [5]. Preliminary material needed to support these query transformation specifications may be found in [4].

## 2 THE TRANSFORMATIONS

Exceptions are raised by the SSQL transformations:

SML

```
exception internalError;  
exception wrongType;  
exception wrongWorth;  
exception notTrigger;  
exception onlyInTriggers;  
exception notMonadic;  
exception notDyadic;  
exception notTriadic;  
exception notSetFunction;  
exception noSuchColumn;  
exception noSuchTable;  
exception noSuchDirectory;  
exception noSuchParameter;  
exception noScope;  
exception wrongScope;  
exception ambiguousName;  
exception tooWide;  
exception emptyUnionList;
```

### 2.1 Incompletely Specified Transformations

The following functions have not been specified in [5]. The exception *notDefined*, from [4], is raised.

SML

```
|fun (check_enum : Enum * int * Table_specssql -> bool) x
|      = raise notDefined "check_enum";
```

SML

```
|fun (check_fixed : Fixed * int * int -> bool) x = raise notDefined "check_fixed";
```

SML

```
|fun (check_floating : Floating * int * int * int -> bool) x
|      = raise notDefined "check_floating";
```

SML

```
|fun (check_interval : Interval * string -> bool) x = raise notDefined "check_interval";
```

SML

```
|fun (check_time : Time * string -> bool) x = raise notDefined "check_time";
```

SML

```
|fun (timeFormatToInterval : string -> string) x
|      = raise notDefined "timeFormatToInterval";
```

SML

```
|fun (unique_name : unit -> string) () = raise notDefined "unique_name";
```

SML

```
|fun (client_clearance : unit -> Class) () = raise notDefined "client_clearance";
```

SML

```
|fun (contextual_data : string -> Constant_valuessql * Class) s
|      = raise notDefined "contextual_data";
```

SML

```
|fun (query_class : unit -> Class) () = raise notDefined "query_class";
```

SML

```
|fun (query_constants_class : unit -> Class) ()
|      = raise notDefined "query_constants_class";
```

SML

```
|fun (default_directory : unit -> string list) () = raise notDefined "default_directory";
```

## 2.2 Symbol Table Model

SML

```

| fun (findcolumn : ColumnSpecification * TableDetail list ->
      (TableDetail * SsqlCol * TsqlCol)list)
      (cs,[]) = []
| findcolumn (cs,tdl) =
  let fun (look : string * TableDetail * SsqlCol list * TsqlCol list
          ->(TableDetail * SsqlCol * TsqlCol)list)
        (n,td,[],x) = []
        | look (n,td,x,[]) = []
        | look (n,td,scl,tcl) =
          let val scs = rev(tl(rev scl))
              val sc = hd(rev scl)
              val tcs = rev(tl(rev tcl))
              val tc = hd(rev tcl)
          in look(n,td,scs,tcs) @
            (if #name sc = names n then [(td,sc,tc)]
             else [])
          end
        val tds = rev(tl(rev tdl))
        val td = hd(rev tdl)
  in findcolumn(cs,tds) @
    (case cs of
      anonymous_column col =>
        look(col,td,#columns td,#implementation td)
    | specific(ts,col) =>
      (case (#tableName td) of
        nametn ts =>
          look(col,td,#columns td,#implementation td)
        | other => []))
    end;

```

SML

```

fun (findident : ColumnSpecification * IdentDetail list -> IdentDetail list)
  (cs,[]) = []
| findident (specific(t,c),ids) = []
| findident (anonymous_column col,idl) =
  let val ids = rev(tl(rev idl))
      val id = hd(rev idl)
  in findident(anonymous_column col,ids) @
    (if col = #identName id
     then [id]
     else [])
  end;

```

SML

```

local fun (look : ColumnSpecification * Scope list -> TableInfo * SsqlCol)
  (cs,[]) = raise noSuchColumn
| look (cs,sl) = let val outer = rev(tl(rev sl))
                  val {tables = t,identifiers = i} = hd(rev sl)
                in case findcolumn(cs,t) of
                    [] => look(cs,outer)
                    [(td,sc,tc)] => (#info td,sc)
                    xs => raise ambiguousName
                end
in
fun (lookupcolumn_info : ColumnSpecification -> TableInfo * SsqlCol)
  cs = look(cs,!symbolTable)
end;

```

SML

```

fun (maxBound : BoundInfo -> Class)
  (upb c) = c
| maxBound (constant c) = c;

```

SML

```

fun (innermost : Scope list -> Scope list)
  [] = []
| innermost sl = let val outer = rev(tl(rev sl))
                    val inner = hd(rev sl)
                    val {tables = tds,identifiers = ids} = inner
                in
                    if tds = [] then innermost outer @ [{tables = [],identifiers = ids}]
                    else [inner]
                end;

```

SML

```

local fun (look : ColumnSpecification * Scope list -> TsqlRepr * Class)
      (cs,[]) = raise noSuchColumn
    | look (cs,sl) =
      let val outer = rev(tl(rev sl))
          val {tables = t,identifiers = i} = hd(rev sl)
      in case (find_column(cs,t),find_ident(cs,i)) of
          ([,[]) => look(cs,outer)
        | ((td,sc,tc),[]) =>
          let val u = maxBound(#col_class sc)
          in case (#class_name tc) of
              anon_tc => raise internalError
            | name_tc s => (column(#genCorr td,s),u)
            | constant_tc c => (constant_class c,u)
          end
        | ([,[id]) =>
          (case (#cName id) of
              none_t => (constant_class (#lub_id id),#lub_id id)
            | anon_t => raise internalError
            | name_t s => (local_identifier s,#lub_id id))
        | (xs,ys) => raise ambiguousName
      end
    in
fun (lookup_col_spec_class : bool * ColumnSpecification -> TsqlRepr * Class)
      (true,cs) = look(cs,innermost (!symbolTable))
    | lookup_col_spec_class (false,cs) = look(cs,!symbolTable)
end;

```

SML

```

local fun (look : ColumnSpecification * Scope list -> TsqlRepr)
      (cs,[]) = raise noSuchColumn
    | look (cs,sl) =
      let val outer = rev(tl(rev sl))
          val {tables = t, identifiers = i} = hd(rev sl)
      in case (find_column(cs,t), find_ident(cs,i)) of
          ([,[]) => look(cs,outer)
        | ((td,sc,tc),[]) =>
            (case (#dinary_name tc) of
              none_t => raise internalError
            | anon_t => raise internalError
            | name_t s => column(#genCorr td,s))
        | ([,id]) =>
            (case (#info id) of
              (st,dinary) =>
                (local_identifier(#vName id))
            | (st,other) => constant_null)
        | (xs,ys) => raise ambiguousName
      end

in
fun (lookup_col_spec_dinary : bool * ColumnSpecification -> TsqlRepr)
      (true,cs) = look(cs,innermost (!symbolTable))
    | lookup_col_spec_dinary (false,cs) = look(cs,!symbolTable)
end;

```

SML

```

local fun (look : ColumnSpecification * Scope list -> TsqlRepr)
      (cs,[]) = raise noSuchColumn
    | look (cs,sl) =
      let val outer = rev(tl(rev sl))
          val {tables = t,identifiers = i} = hd(rev sl)
      in case (find_column(cs,t),find_ident(cs,i)) of
          ([,[]) => look(cs,outer)
        | ((td,sc,tc),[]) =>
            (case (#dinary_name tc) of
              none_t => raise internalError
              | anon_t => raise internalError
              | name_t s => column(#genCorr td,s))
          | ([,[id]) =>
            (case (#info id) of
              (st,sterling) =>
                (local_identifier(#vName id))
              | (st,other) => constant_null)
          | (xs,ys) => raise ambiguousName
          end
      in
fun (lookup_col_spec_sterling : bool * ColumnSpecification -> TsqlRepr)
      (true,cs) = look(cs,innermost (!symbolTable))
    | lookup_col_spec_sterling (false,cs) = look(cs,!symbolTable)
end;

```

SML

```

local fun (project_implementation : TableDetail -> TsqlCol list)
      td = #implementation td
    fun (extract_implementation : Scope -> TsqlCol list)
      sc = fold (op @) (map project_implementation (#tables sc))
    in
fun (lookup_local_col_implementation : unit -> TsqlCol list)
      () = case (fold (op @)(map extract_implementation(innermost (!symbolTable)))) of
          [] => raise noScope
        | trs => trs
    end;

```

SML

```

local fun (project_columns : TableDetail -> SsqlCol list)
      td = #columns td
      fun (extract_columns : Scope -> SsqlCol list)
            sc = fold (op @) (map project_columns (#tables sc))
in
fun (lookup_local_col_info : unit -> SsqlCol list)
  () = case (fold (op @)(map extract_columns(innermost (!symbolTable)))) of
        [] => raise noScope
        | trs => trs
end;

```

SML

```

local fun (look2 : string * SsqlCol * TsqlCol -> TsqlRepr * Class)
      (corr,sc,tc) = let val u = maxBound(#col_class sc)
                    in case (#class_name tc) of
                        anon_tc => raise internalError
                        | name_tc s => (column(corr,s),u)
                        | constant_tc c => (constant_class c,u)
                    end
      fun (look1 : TableDetail -> (TsqlRepr * Class) list)
            td = at3 (map look2)(seq(length(#columns td),#genCorr td),
                                #columns td,#implementation td)
      fun (look : Scope -> (TsqlRepr * Class) list)
            sc = fold (op @) (map look1 (#tables sc))
in
fun (lookup_local_col_spec_classes : unit -> (TsqlRepr * Class) list)
  () = case (fold (op @)(map look (innermost (!symbolTable)))) of
        [] => raise noScope
        | trs => trs
end;

```

SML

```

local fun (look2 : string * TsqlCol -> TsqlRepr)
      (corr,tc) = case (#sterling_name tc) of
                  none_t => constant_null
                  | anon_t => raise internalError
                  | name_t s => column(corr,s)
fun (look1 : TableDetail -> TsqlRepr list)
      td = at2 (map look2)(seq(length(#columns td),#genCorr td),
              #implementation td)
fun (look : Scope -> TsqlRepr list)
      sc = fold (op @) (map look1 (#tables sc))
in
fun (lookuplocal_col_spec_sterlings : unit -> TsqlRepr list)
      () = case (fold (op @)(map look (innermost (!symbolTable)))) of
            [] => raise noScope
            | trs => trs
end;

```

SML

```

local fun (look1 : TableDetail -> TsqlRepr)
      td = case (#rowClass td) of
            anontc => raise internalError
            | nametc s => column(#genCorr td,s)
            | constanttc c => constant_class c
fun (look : Scope -> TsqlRepr list)
      sc = (map look1 (#tables sc))
in
fun (lookuplocal_row_classes : unit -> TsqlRepr list)
      () = case (fold (op @)(map look (innermost (!symbolTable)))) of
            [] => raise noScope
            | trs => trs
end;

```

SML

```

local fun (look : ColumnSpecification * Scope list -> TsqlRepr)
      (cs,[]) = raise noSuchColumn
    | look (cs,sl) =
      let val outer = rev(tl(rev sl))
          val {tables = t, identifiers = i} = hd(rev sl)
      in case (find_column(cs,t), find_ident(cs,i)) of
          ([,[]) => look(cs,outer)
        | ((td,sc,tc),[]) =>
          (case (#rowClass td) of
            anontc => raise internalError
          | nametc s => column(#genCorr td,s)
          | constanttc c => constant_class c)
        | ([, [id]) => constant_class (query_class())
        | (xs,ys) => raise ambiguousName
      end
    in
      fun (lookupcolumn_row_class : bool * ColumnSpecification -> TsqlRepr)
            (true,cs) = look(cs,innermost (!symbolTable))
          | lookupcolumn_row_class (false,cs) = look(cs,!symbolTable)
      end;

```

SML

```

local fun (look1 : TableSpecification * TableDetail -> TsqlRepr list)
      (ts,td) = case (#tableName td) of
                anontn => []
                | nametn tn => if ts = tn then
                              (case (#rowClass td) of
                                anontc => raise internalError
                                | nametc s =>
                                  [column(#genCorr td,s)]
                                | constanttc c =>
                                  [constant_class c])
                              else []
      fun (look : TableSpecification * Scope list -> TsqlRepr)
        (ts,[]) = raise noSuchTable
        | look (ts,sl) =
          let val outer = rev(tl(rev sl))
              val {tables = tds,identifiers = ids} = hd(rev sl)
          in (case fold(op @)(at2 (map look1)(seq(length tds,ts),tds)) of
              [] => look(ts,outer)
              | [tr] => tr
              | trs => raise ambiguousName)
          end
in
fun (lookuptable_row_class : bool * TableSpecification -> TsqlRepr)
  (true,ts) = look(ts,innermost (!symbolTable))
| lookuptable_row_class (false,ts) = look(ts,!symbolTable)
end;

```

SML

```

local fun (look1 : TableSpecification * TableDetail -> TableDetail list)
      (ts,td) = case (#tableName td) of
                anontn => []
                | nametn tn => if ts = tn then [td]
                               else []
      fun (look : TableSpecification * Scope list -> TableDetail)
          (ts,[]) = raise noSuchTable
          | look (ts,sl) =
            let val outer = rev(tl(rev sl))
                val {tables = tds,identifiers = ids} = hd(rev sl)
            in (case fold(op @)(at2 (map look1)(seq(length tds,ts),tds)) of
                [] => look(ts,outer)
                | [ti] => ti
                | tis => raise ambiguousName)
            end
in
fun (lookuptable_detail : TableSpecification -> TableDetail)
  ts = look(ts,!symbolTable)
end;

```

## 2.3 Symbol Table Operations

SML

```

local fun (find : string * IdentDetail -> IdentDetail list)
      (n,id) =      if n = #identName id
                    then [id]
                    else []
in
fun (enteridentifier : string * ExpType * Class -> string * string)
  (name,et,up) =
  case !symbolTable of
    [] => raise noScope
  | sl => let   val outer = rev(tl(rev sl))
              val {tables = tds,identifiers = ids} = hd(rev sl)
              in  if tds = [] then
                  let   val unv = unique_name()
                        val unc = unique_name()
                        val id = {identName = name,
                                info = et,
                                lubid = up,
                                vName = unv,
                                cName = namet unc}
                  in   if at2(map find) (seq(length ids,name),ids) <> []
                      then raise ambiguousName
                      else
                        let val side_effect = symbolTable := outer @
                                                [{tables = [],identifiers = ids @ [id]}]
                        in (unv,unc)
                        end
                  end
              else raise wrongScope
          end
end;

```

SML

```

local fun (find : string * IdentDetail -> IdentDetail list)
      (n,id) = if n = #identName id
              then [id]
              else []
in
fun (enter_identifier_constant_class : string * ExpType * Class -> string)
  (name,et,clasf) =
  case !symbolTable of
    [] => raise noScope
  | sl => let val outer = rev(tl(rev sl))
          val {tables = tds,identifiers = ids} = hd(rev sl)
          in if tds = [] then
              let val unv = unique_name()
                  val id = {identName = name,
                           info = et,
                           lub_id = clasf,
                           vName = unv,
                           cName = none_t}
              in if at2(map find) (seq(length ids,name),ids) <> []
                 then raise ambiguousName
                 else
                  let val side_effect = symbolTable := outer @
                                         [{tables = [],identifiers = ids @ [id]}]
                  in unv
                  end
              end
          else raise wrongScope
          end
end;

```

SML

```

fun (extract_parameter : string * ParamInfo list -> ParamInfo list)
  (name,[]) = []
| extract_parameter (name,l) = let val more = rev(tl(rev l))
                                val pi = hd(rev l)
                                in
                                  if name = #name pi
                                  then extract_parameter(name,more) @ [pi]
                                  else extract_parameter(name,more)
                                end;

```

SML

```

fun (enter_parameter : string * Constant_valuessql * Class -> unit) (name,v,clasf) =
  if !symbolTable <> []
  then raise wrongScope
  else case extract_parameter(name,!parameterTable) of
        [] => parameterTable := !parameterTable @
            [{name =name,valp =v,clasf =clasf}]
        | ps => raise ambiguousName;

```

SML

```

fun (enter_corr_table : string * TableName * TableInfo * SsqlCol list * TsqlClassName
    * TsqlCol list -> string)
  (cn,ts,ti,scs,rcn,tcs) =
  case !symbolTable of
    [] => raise noScope
  | sl => let    val outer = rev(tl(rev sl))
                val {tables = tds,identifiers = ids} = hd(rev sl)
            in   if ids = [] then
                let    val gc = unique_name()
                    val td = {tableName = ts,
                              corrName = names cn,
                              genCorr = gc,
                              info = ti,
                              columns = scs,
                              rowClass = rcn,
                              implementation = tcs,
                              constraints = {null_allowed=[],
                                             lwb=[],
                                             unique=[],
                                             uniform=[],
                                             index=[]}}
                in   val side_effect = symbolTable := outer @
                    [{tables = rev(td::tds),identifiers=[]}]
                end
            else raise wrongScope
  end;

```

SML

```

fun (entertable : TableName * TableInfo * SsqlCol list
      * TsqlClassName * TsqlCol list -> string)
  (ts,ti,scs,rcn,tcs) =
  case !symbolTable of
    [] => raise noScope
  | sl => let    val outer = rev(tl(rev sl))
              val {tables = tds,identifiers = ids} = hd(rev sl)
              in  if ids = [] then
                  let    val gc = unique_name()
                      val td = {tableName = ts,
                                corrName = anon_s,
                                genCorr = gc,
                                info = ti,
                                columns = scs,
                                rowClass = rcn,
                                implementation = tcs,
                                constraints = {null_allowed=[],
                                              lwb=[],
                                              unique=[],
                                              uniform=[],
                                              index=[]}}
                      val side_effect = symbolTable := outer @
                                         [{tables = rev(td::tds),identifiers=[]}]
                      in gc
                      end
                  else raise wrongScope
              end;
end;

```

SML

```

fun (enter_scope : unit -> unit) () =
  symbolTable := !symbolTable @ [{tables = [],identifiers = []}];

```

SML

```

fun (leave_scope : unit -> unit) () =
  case !symbolTable of
    [] => raise noScope
  | sl => symbolTable := rev(tl(rev sl));

```

SML

```

fun (gettable_info : TableSpecification -> TableInfo * ConstraintInfo * SsqlCol list
      * TsqlClassName * TsqlCol list) x
  = raise notDefined "gettable_info";

```

SML

```
| fun (lookuplocal_table_implementation : unit -> TsqlClassName list ) ()
|                                     = raise notDefined "lookuplocal_table_implementation";
```

SML

```
| fun (lookuplocal_table_info : unit -> TableInfo list )
| () = let fun (look1 : TableDetail -> TableInfo)
|         td = #info td
|         fun (look : Scope -> TableInfo list)
|             sc = map look1 (#tables sc)
|
|         in
|         case fold(op @)(map look(innermost (!symbolTable))) of
|             [] => raise noScope
|             | trs => trs
|         end;
```

SML

```
| fun (lookupparam_data : string -> Constant_valuessql * Class)
| name = case extractparameter(name,!parameterTable) of
|         [] => raise noSuchParameter
|         [info] => (#valp info,#clasf info)
|         other => raise internalError;
```

Entering and leaving scope is achieved by side effect. We define a function to this.

SML

```
| fun (in_new_scope (what : unit -> 'a):'a) = (
| let val side_effect = enter_scope()
|     val body = what()
|     val side_effect = leave_scope()
| in body
| end);
```

## 2.4 Transformations Proper

SML

```
| fun (repr_col : TsqlRepr -> Col_spectsql)
|     (local_identifier name) = denote_col_spect name
| | repr_col (column(corr,col)) = absolute_col_spect([],corr,col)
| | repr_col (constant_class c) = raise internalError
| | repr_col constant_null = raise internalError;
```

SML

```

| fun (all_data_columnslocal : unit -> Col_spectsql list) ()
|   = map repr_col(lookuplocal_col_spec_sterlings());

```

SML

```

| fun (binop_type : Op * SwordType * SwordType -> SwordType)
|   (plus_op, fixedType(p1, s1), fixedType(p2, s2)) =
|     fixedType(p1 max p2, s1 max s2)
|
|   binop_type (plus_op, floatingType(m1, e1, os1), floatingType(m2, e2, os2)) =
|     if m1 = m2 andalso e1 = e2 andalso os1 = os2
|     then floatingType(m1, e1, os1)
|     else raise wrongType
|
|   binop_type (plus_op, enumType(p1, t1), fixedType(p2, 0)) = enumType(p1, t1)
|   binop_type (plus_op, timeType f1, intervalType f2) = timeType f1
|   binop_type (plus_op, intervalType f1, intervalType f2) =
|     if f1 = f2
|     then intervalType f1
|     else raise wrongType
|
|   binop_type (plus_op, t1, t2) = raise wrongType

```

SML

```

|   binop_type (minusd_op, fixedType(p1, s1), fixedType(p2, s2)) =
|     fixedType(p1 max p2, s1 max s2)
|
|   binop_type (minusd_op, floatingType(m1, e1, os1), floatingType(m2, e2, os2)) =
|     if m1 = m2 andalso e1 = e2 andalso os1 = os2
|     then floatingType(m1, e1, os1)
|     else raise wrongType
|
|   binop_type (minusd_op, enumType(p1, t1), fixedType(p2, 0)) = enumType(p1, t1)
|   binop_type (minusd_op, timeType f1, intervalType f2) = timeType f1
|   binop_type (minusd_op, intervalType f1, intervalType f2) =
|     if f1 = f2
|     then intervalType f1
|     else raise wrongType
|
|   binop_type (minusd_op, t1, t2) = raise wrongType

```

SML

```

|      binop_type      (times_op, fixedType(p1,s1), fixedType(p2,s2)) =
|                      fixedType(p1 max p2, s1 max s2)
|
|      binop_type      (times_op, floatingType(m1,e1,os1), floatingType(m2,e2,os2)) =
|                      if m1 = m2 andalso e1 = e2 andalso os1 = os2
|                      then floatingType(m1,e1,os1)
|                      else raise wrongType
|
|      binop_type      (times_op, intervalType f, fixedType(p,s)) = intervalType f
|      binop_type      (times_op, intervalType f, floatingType(m,e,os)) = intervalType f
|      binop_type      (times_op, t1, t2) = raise wrongType

```

SML

```

|      binop_type      (divide_op, fixedType(p1,s1), fixedType(p2,s2)) =
|                      fixedType(p1 max p2, s1 max s2)
|
|      binop_type      (divide_op, floatingType(m1,e1,os1), floatingType(m2,e2,os2)) =
|                      if m1 = m2 andalso e1 = e2 andalso os1 = os2
|                      then floatingType(m1,e1,os1)
|                      else raise wrongType
|
|      binop_type      (divide_op, intervalType f, fixedType(p,s)) = intervalType f
|      binop_type      (divide_op, intervalType f, floatingType(m,e,os)) = intervalType f
|      binop_type      (divide_op, t1, t2) = raise wrongType

```

SML

```

|      binop_type      (concat_op, stringType(min1,max1), stringType(min2,max2)) =
|                      stringType(min1 + min2, max1 + max2)
|
|      binop_type      (concat_op, t1, t2) = raise wrongType

```

SML

```

|      binop_type      (and_op, booleanType, booleanType) = booleanType
|      binop_type      (and_op, t1, t2) = raise wrongType

```

SML

```

|      binop_type      (or_op, booleanType, booleanType) = booleanType
|      binop_type      (or_op, t1, t2) = raise wrongType

```

SML

```

| binop_type (less_than_op,stringType(min1,max1),stringType(min2,max2)) =
|             booleanType
| binop_type (less_than_op,fixedType(p1,s1),fixedType(p2,s2)) =
|             booleanType
| binop_type (less_than_op,floatingType(m1,e1,os1),floatingType(m2,e2,os2)) =
|             booleanType
| binop_type (less_than_op,enumType(p1,t1),enumType(p2,t2)) =
|             if p1 = p2 andalso t1 = t2
|             then booleanType
|             else raise wrongType
| binop_type (less_than_op,timeType f1,timeType f2) =
|             booleanType
| binop_type (less_than_op,intervalType f1,intervalType f2) =
|             booleanType
| binop_type (less_than_op,t1,t2) = raise wrongType

```

SML

```

| binop_type (less_or_equal_op,stringType(min1,max1),
|             stringType(min2,max2)) =
|             booleanType
| binop_type (less_or_equal_op,fixedType(p1,s1),fixedType(p2,s2)) =
|             booleanType
| binop_type (less_or_equal_op,floatingType(m1,e1,os1),
|             floatingType(m2,e2,os2)) =
|             booleanType
| binop_type (less_or_equal_op,enumType(p1,t1),enumType(p2,t2)) =
|             if p1 = p2 andalso t1 = t2
|             then booleanType
|             else raise wrongType
| binop_type (less_or_equal_op,timeType f1,timeType f2) =
|             booleanType
| binop_type (less_or_equal_op,intervalType f1,intervalType f2) =
|             booleanType
| binop_type (less_or_equal_op,t1,t2) = raise wrongType

```

SML

```

|   binop_type      (greater_or_equal_op,stringType(min1,max1),
|                   stringType(min2,max2)) =
|                   booleanType
|
|   binop_type      (greater_or_equal_op,fixedType(p1,s1),fixedType(p2,s2)) =
|                   booleanType
|
|   binop_type      (greater_or_equal_op,floatingType(m1,e1,os1),
|                   floatingType(m2,e2,os2)) =
|                   booleanType
|
|   binop_type      (greater_or_equal_op,enumType(p1,t1),enumType(p2,t2)) =
|                   if p1 = p2 andalso t1 = t2
|                   then booleanType
|                   else raise wrongType
|
|   binop_type      (greater_or_equal_op,timeType f1,timeType f2) =
|                   booleanType
|
|   binop_type      (greater_or_equal_op,intervalType f1,intervalType f2) =
|                   booleanType
|
|   binop_type      (greater_or_equal_op,t1,t2) = raise wrongType

```

SML

```

|   binop_type      (greater_than_op,stringType(min1,max1),
|                   stringType(min2,max2)) =
|                   booleanType
|
|   binop_type      (greater_than_op,fixedType(p1,s1),fixedType(p2,s2)) =
|                   booleanType
|
|   binop_type      (greater_than_op,floatingType(m1,e1,os1),
|                   floatingType(m2,e2,os2)) =
|                   booleanType
|
|   binop_type      (greater_than_op,enumType(p1,t1),enumType(p2,t2)) =
|                   if p1 = p2 andalso t1 = t2
|                   then booleanType
|                   else raise wrongType
|
|   binop_type      (greater_than_op,timeType f1,timeType f2) =
|                   booleanType
|
|   binop_type      (greater_than_op,intervalType f1,intervalType f2) =
|                   booleanType
|
|   binop_type      (greater_than_op,t1,t2) = raise wrongType

```

SML

```

| binop_type (equal_op,nullType,t) = booleanType
| binop_type (equal_op,t,nullType) = booleanType
| binop_type (equal_op,booleanType,booleanType) = booleanType
| binop_type (equal_op,stringType(min1,max1),stringType(min2,max2)) =
| booleanType
| binop_type (equal_op,fixedType(p1,s1),fixedType(p2,s2)) =
| booleanType
| binop_type (equal_op,enumType(p1,t1),enumType(p2,t2)) =
| if p1 = p2 andalso t1 = t2
| then booleanType
| else raise wrongType
| binop_type (equal_op,floatingType(m1,e1,os1),floatingType(m2,e2,os2)) =
| booleanType
| binop_type (equal_op,timeType f1,timeType f2) = booleanType
| binop_type (equal_op,intervalType f1,intervalType f2) = booleanType
| binop_type (equal_op,classType,classType) = booleanType
| binop_type (equal_op,codeType,codeType) = booleanType
| binop_type (equal_op,anyType,anyType) = booleanType
| binop_type (equal_op,t1,t2) = raise wrongType

```

SML

```

| binop_type (not_equal_op,nullType,t) = booleanType
| binop_type (not_equal_op,t,nullType) = booleanType
| binop_type (not_equal_op,booleanType,booleanType) = booleanType
| binop_type (not_equal_op,stringType(min1,max1),stringType(min2,max2)) =
| booleanType
| binop_type (not_equal_op,fixedType(p1,s1),fixedType(p2,s2)) =
| booleanType
| binop_type (not_equal_op,enumType(p1,t1),enumType(p2,t2)) =
| if p1 = p2 andalso t1 = t2
| then booleanType
| else raise wrongType
| binop_type (not_equal_op,floatingType(m1,e1,os1),
| floatingType(m2,e2,os2)) =
| booleanType
| binop_type (not_equal_op,timeType f1,timeType f2) = booleanType
| binop_type (not_equal_op,intervalType f1,intervalType f2) = booleanType
| binop_type (not_equal_op,classType,classType) = booleanType
| binop_type (not_equal_op,codeType,codeType) = booleanType
| binop_type (not_equal_op,anyType,anyType) = booleanType
| binop_type (not_equal_op,t1,t2) = raise wrongType

```

SML

```

| | binop_type (lub_op,classType,classType) = classType
| | binop_type (lub_op,t1,t2) = raise wrongType

```

SML

```

| | binop_type (glb_op,classType,classType) = classType
| | binop_type (glb_op,t1,t2) = raise wrongType

```

SML

```

| | binop_type (dom_op,classType,classType) = booleanType
| | binop_type (dom_op,t1,t2) = raise wrongType

```

SML

```

| | binop_type (dom_by_op,classType,classType) = booleanType
| | binop_type (dom_by_op,t1,t2) = raise wrongType

```

SML

```

| | binop_type (liked_op,stringType(min1,max1),stringType(min2,max2)) =
| | booleanType
| | binop_type (liked_op,t1,t2) = raise wrongType

```

SML

```

| | binop_type (maximum_op,fixedType(p1,s1),fixedType(p2,s2)) =
| | fixedType(p1 max p2,s1 max s2)
| | binop_type (maximum_op,enumType(p1,t1),enumType(p2,t2)) =
| | if p1 = p2 andalso t1 = t2
| | then enumType(p1,t1)
| | else raise wrongType
| | binop_type (maximum_op,floatingType(m1,e1,os1),
| | floatingType(m2,e2,os2)) =
| | if m1 = m2 andalso e1 = e2 andalso os1 = os2
| | then floatingType(m1,e1,os1)
| | else raise wrongType
| | binop_type (maximum_op,timeType f1,timeType f2) =
| | if f1 = f2
| | then timeType f1
| | else raise wrongType
| | binop_type (maximum_op,intervalType f1,intervalType f2) =
| | if f1 = f2
| | then intervalType f1
| | else raise wrongType
| | binop_type (maximum_op,t1,t2) = raise wrongType

```

SML

```

| | binop_type (minimum_op, fixedType(p1,s1), fixedType(p2,s2)) =
| |           fixedType(p1 max p2, s1 max s2)
| | binop_type (minimum_op, enumType(p1,t1), enumType(p2,t2)) =
| |           if p1 = p2 andalso t1 = t2
| |           then enumType(p1,t1)
| |           else raise wrongType
| | binop_type (minimum_op, floatingType(m1,e1,os1),
| |           floatingType(m2,e2,os2)) =
| |           if m1 = m2 andalso e1 = e2 andalso os1 = os2
| |           then floatingType(m1,e1,os1)
| |           else raise wrongType
| | binop_type (minimum_op, timeType f1, timeType f2) =
| |           if f1 = f2
| |           then timeType f1
| |           else raise wrongType
| | binop_type (minimum_op, intervalType f1, intervalType f2) =
| |           if f1 = f2
| |           then intervalType f1
| |           else raise wrongType
| | binop_type (minimum_op, t1, t2) = raise wrongType

```

SML

```

| | binop_type (opr, t1, t2) = raise notDyadic;

```

SML

```

| fun (monop_type : Op * SwordType -> SwordType)
|   (not_op, booleanType) = booleanType
|   monop_type (not_op, t) = raise wrongType

```

SML

```

| | monop_type (definitely_op, booleanType) = booleanType
| | monop_type (definitely_op, t) = raise wrongType

```

SML

```

| | monop_type (possibly_op, booleanType) = booleanType
| | monop_type (possibly_op, t) = raise wrongType

```

SML

```

| | monop_type (minus_op, fixedType(p,s)) = fixedType(p,s)
| | monop_type (minus_op, floatingType(m,e,os)) = floatingType(m,e,os)
| | monop_type (minus_op, intervalType f) = intervalType f
| | monop_type (minus_op, t) = raise wrongType

```

SML

```

| |   monop_type   (ord_op,enumType(p,t)) = fixedType(p,0)
| |   monop_type   (ord_op,stringType(mini,maxi)) = fixedType(1,0)
| |   monop_type   (ord_op,t) = raise wrongType

```

SML

```

| |   monop_type   (char_op,fixedType(p,s)) = stringType(1,1)
| |   monop_type   (char_op,t) = raise wrongType

```

SML

```

| |   monop_type   (upper_op,stringType(mini,maxi)) = stringType(mini,maxi)
| |   monop_type   (upper_op,t) = raise wrongType

```

SML

```

| |   monop_type   (lower_op,stringType(mini,maxi)) = stringType(mini,maxi)
| |   monop_type   (lower_op,t) = raise wrongType

```

SML

```

| |   monop_type   (opr,t) = raise notMonadic;

```

SML

```

| fun   (triop_type : Op * SwordType * SwordType * SwordType -> SwordType)
|       (liket_op,stringType(min1,max1),stringType(min2,max2),
|       stringType(min3,max3)) = booleanType
|
|   triop_type   (liket_op,t1,t2,t3) = raise wrongType
|   triop_type   (between_op,fixedType(p1,s1),fixedType(p2,s2),
|       fixedType(p3,s3)) = booleanType
|   triop_type   (between_op,floatingType(m1,e1,os1),floatingType(m2,e2,os2),
|       floatingType(m3,e3,os3)) = booleanType
|   triop_type   (between_op,stringType(min1,max1),stringType(min2,max2),
|       stringType(min3,max3)) = booleanType
|   triop_type   (between_op,enumType(p1,t1),enumType(p2,t2),enumType(p3,t3)) =
|       if p1 = p2 andalso p2 = p3 andalso t1 = t2 andalso t2 = t3
|       then booleanType
|       else raise wrongType
|   triop_type   (between_op,timeType(f1),timeType(f2),
|       timeType(f3)) = booleanType
|   triop_type   (between_op,intervalType(f1),intervalType(f2),
|       intervalType(f3)) = booleanType
|   triop_type   (between_op,classType,classType,classType) = booleanType
|   triop_type   (between_op,t1,t2,t3) = raise wrongType
|   triop_type   (opr,t1,t2,t3) = raise notTriadic;

```

SML

```

|fun  (set_func_type : Op * SwordType -> SwordType)
|      (plus_op, fixedType(p,s)) = fixedType(p,s)
|      set_func_type             (plus_op, floatingType(m,e,os)) = floatingType(m,e,os)
|      set_func_type             (plus_op, intervalType f) = intervalType f
|      set_func_type             (plus_op, t) = raise wrongType

```

SML

```

|      set_func_type             (concat_op, stringType (mini,maxi)) = stringType (mini,maxi)
|      set_func_type             (concat_op, t) = raise wrongType

```

SML

```

|      set_func_type             (and_op, booleanType) = booleanType
|      set_func_type             (and_op, t) = raise wrongType

```

SML

```

|      set_func_type             (or_op, booleanType) = booleanType
|      set_func_type             (or_op, t) = raise wrongType

```

SML

```

|      set_func_type             (lub_op, classType) = classType
|      set_func_type             (lub_op, t) = raise wrongType

```

SML

```

|      set_func_type             (glb_op, classType) = classType
|      set_func_type             (glb_op, t) = raise wrongType

```

SML

```

|      set_func_type             (maximum_op, fixedType(p,s)) = fixedType(p,s)
|      set_func_type             (maximum_op, floatingType(m,e,os)) = floatingType(m,e,os)
|      set_func_type             (maximum_op, enumType(p,t)) = enumType(p,t)
|      set_func_type             (maximum_op, intervalType f) =
|      intervalType f
|      set_func_type             (maximum_op, timeType f) =
|      timeType f
|      set_func_type             (maximum_op, t) = raise wrongType

```

SML

```

|      set_func_type             (minimum_op, fixedType(p,s)) = fixedType(p,s)
|      set_func_type             (minimum_op, floatingType(m,e,os)) = floatingType(m,e,os)
|      set_func_type             (minimum_op, enumType(p,t)) = enumType(p,t)
|      set_func_type             (minimum_op, intervalType f) =
|      intervalType f
|      set_func_type             (minimum_op, timeType f) = timeType f
|      set_func_type             (minimum_op, t) = raise wrongType

```

SML

```

| |   set_func_type      (average_op, fixedType(p,s)) = fixedType(p,s)
| |   set_func_type      (average_op, floatingType(m,e,os)) = floatingType(m,e,os)
| |   set_func_type      (average_op, intervalType f) =
| |   intervalType f
| |   set_func_type      (average_op,t) = raise wrongType;

```

SML

```

| fun (check_boolean : ExpType -> ExpType) (booleanType,w) = (booleanType,w)
| |   check_boolean      (t,w) = raise wrongType;

```

SML

```

| fun (check_type_conversion : SwordType * SwordType -> unit)
| |   (nullType,t) = ()
| |   check_type_conversion(stringType(min1,max1),stringType(min2,max2)) = ()
| |   check_type_conversion(stringType(mini,maxi),fixedType(p,s)) = ()
| |   check_type_conversion(stringType(mini,maxi),floatingType(m,e,os)) = ()
| |   check_type_conversion(stringType(mini,maxi),timeType f) = ()
| |   check_type_conversion(stringType(mini,maxi),intervalType f) = ()
| |   check_type_conversion(stringType(mini,maxi),enumType(p,t)) = ()
| |   check_type_conversion(stringType(mini,maxi),codeType) = ()

```

SML

```

| |   check_type_conversion(fixedType(p,s),stringType(mini,maxi)) = ()
| |   check_type_conversion(fixedType(p1,s1),fixedType(p2,s2)) = ()
| |   check_type_conversion(fixedType(p,s),floatingType(m,e,os)) = ()

```

SML

```

| |   check_type_conversion(floatingType(m,e,os),stringType(mini,maxi)) = ()
| |   check_type_conversion(floatingType(m,e,os),fixedType(p,s)) = ()
| |   check_type_conversion(floatingType(m1,e1,os1),floatingType(m2,e2,os2)) = ()

```

SML

```

| |   check_type_conversion(timeType f,stringType(mini,maxi)) = ()
| |   check_type_conversion(intervalType f,stringType(mini,maxi)) = ()

```

SML

```

| |   check_type_conversion(enumType(p,t),stringType(mini,maxi)) = ()

```

SML

```

| |   check_type_conversion(timeType f1,timeType f2) = ()

```

SML

```
| | check_type_conversion(intervalType f1,intervalType f2) = ()
```

SML

```
| | check_type_conversion(codeType,stringType(mini,maxi)) = ()
```

SML

```
| | check_type_conversion(anyType,t) = ()
```

```
| | check_type_conversion(t,anyType) = ()
```

SML

```
| | check_type_conversion (t1,t2) = raise wrongType;
```

SML

```
| fun (check_type_conversion_domain : SwordType * SwordType -> unit)
| | (stringType(min1,max1),timeType f) = ()
| | check_type_conversion_domain (stringType(min1,max1),classType) = ()
| | check_type_conversion_domain (timeType f,stringType(min1,max1)) = ()
| | check_type_conversion_domain (classType,stringType(min1,max1)) = ()
| | check_type_conversion_domain (t1,t2) = raise wrongType;
```

SML

```
| fun (convert_col_spec : Col_spec_ssql -> ColumnSpecification)
| | (denote_col_spec_s col) = anonymous_column col
| | convert_col_spec (absolute_col_spec_s(dir,tab,col)) =
| | | specific(absolute(dir,tab),col)
| | convert_col_spec (default_col_spec_s(up,dir,tab,col)) =
| | | specific(default(up,dir,tab),col);
```

SML

```
| fun (class_column : Col_spec_ssql-> (Col_spec_tsql,Class)Sum) cs =
| case (lookup_col_spec_class(true,convert_col_spec cs)) of
| | (local_identifier name, lub_cl) => inL(denote_col_spec_t name)
| | (column(gen_corr,gen_col), lub_cl) =>
| | | inL(absolute_col_spec_t([],gen_corr,gen_col))
| | (constant_class cl, lub_cl) => inR cl
| | (constant_null, lub_cl) => raise internalError;
```

SML

```
| fun (denote_name : TsqlRepr -> Value_tsql)
| | (local_identifier s) = contents_t(denote_col_spec_t s)
| | denote_name (column(cn,col)) = contents_t(absolute_col_spec_t([],cn,col))
| | denote_name (constant_class c) = denote_class_t c
| | denote_name constant_null = denote_null_t;
```

SML

```

fun (column_data_test : Col_specssql -> Valuetsql list) cs =
  let val (tr,u) = lookupcol_spec_class(false,convertcol_spec cs)
  in
  if client_clearance() dom u
  then []
  else let val cc = denote_classt(client_clearance())
        in [binopt(dom_op,(cc,denote_name tr))]
        end
  end;
end;

```

SML

```

fun (col_exp : ExpType -> ColType)
  (t,dinary) = (nullType,t)
| col_exp (t,sterling) = (t,nullType)
| col_exp (t,worthless) = (t,nullType)
| col_exp (t,priceless) = raise wrongType;

```

SML

```

fun (col_target : SsqlCol -> TsqlCol) sc =
  let fun (bound : BoundInfo -> TsqlClassName) (upb c) = anontc
        | bound (constant c) = constanttc c
        fun (target : ColType * TsqlClassName -> TsqlCol)
            ((nullType,nullType),c) = raise internalError
        | target ((s,nullType),c) = {sterling_name = anont,
                                     dinary_name = nonet,
                                     class_name = c}
        | target ((nullType,d),c) = {sterling_name = nonet,
                                     dinary_name = anont,
                                     class_name = c}
        | target ((s,d),c) = {sterling_name = anont,
                              dinary_name = anont,
                              class_name = c}
  in target(#type_field sc,bound(#col_class sc))
  end;
end;

```

SML

```

fun (converttable_spec : Table_specssql -> TableSpecification)
  (absolute_table_specs(dir,tab)) = absolute(dir,tab)
| converttable_spec (default_table_specs(up,dir,tab)) = default(up,dir,tab);

```

SML

```

fun (constant_value_type : Constant_valuessql -> SwordType)
  | denote_nulls = nullType
  | constant_value_type denote_voids = monoleanType
  | constant_value_type denote_trues = booleanType
  | constant_value_type denote_falses = booleanType
  | constant_value_type (denote_strings(s,string_types(mini,maxi))) =
    (if mini <= size s andalso size s <= maxi
     then stringType(mini,maxi)
     else raise wrongType)
  | constant_value_type (denote_fixeds(f,fixed_types(p,s))) =
    (if check_fixed(f,p,s)
     then fixedType(p,s)
     else raise wrongType)
  | constant_value_type (denote_floatings(f,floating_types(m,e,os))) =
    (if check_floating(f,m,e,os)
     then floatingType(m,e,os)
     else raise wrongType)
  | constant_value_type (denote_enums(e,enum_types(p,t))) =
    (if check_enum(e,p,t)
     then enumType(p,converttable_spec t)
     else raise wrongType)
  | constant_value_type (denote_times(tm,time_types f)) =
    (if check_time(tm,f)
     then timeType f
     else raise wrongType)
  | constant_value_type (denote_intervals(i,interval_types f)) =
    (if check_interval(i,f)
     then intervalType f
     else raise wrongType)
  | constant_value_type (denote_classs c) = classType
  | constant_value_type (denote_codes c) = codeType
  | constant_value_type other = raise wrongType;

```

SML

```

fun (convertssql_type : Typessql -> SwordType)
    monolean_types = monoleanType
|
|   convertssql_type   boolean_types = booleanType
|   convertssql_type   (string_types(mini,maxi)) = stringType(mini,maxi)
|   convertssql_type   (fixed_types(p,s)) = fixedType(p,s)
|   convertssql_type   (floating_types(m,e,os)) = floatingType(m,e,os)
|   convertssql_type   (enum_types(p,t)) = enumType(p,converttable_spec(t))
|   convertssql_type   (time_types f) = timeType f
|   convertssql_type   (interval_types f) = intervalType f
|   convertssql_type   class_types = classType
|   convertssql_type   code_types = codeType
|   convertssql_type   any_types = anyType;

```

SML

```

fun (converttableSpecification : TableSpecification -> Tablespec_tsql)
    (absolute(directory,table)) = absolutetable_spec_t(directory,table)
|
|   converttableSpecification (default(up,directory,table)) =
|       let fun (backUp : string list * int -> string list)(dir,0) = dir
|           | backUp (dir,n) = if n > 0 andalso length dir > 0
|               then backUp(rev(tl(rev dir)),n - 1)
|               else raise noSuchDirectory
|           val dir = backUp(defaultdirectory(),up) @ directory
|           in absolutetable_spec_t(dir,table)
|       end;

```

SML

```

local fun (dot : string -> string) s = s ^ "."
in
fun (table_name : TableSpecification -> string)
    (absolute([],tab)) = tab
|   table_name (absolute(dir,tab)) = (fold (op ^)(map dot dir)) ^ tab
|   table_name (default(up,[],tab)) = implode(seq(up,"-")) ^ tab
|   table_name (default(up,dir,tab)) = implode(seq(up,"-"))
|       ^ (fold (op ^)(map dot dir)) ^ tab
end;

```

SML

```

fun (convertsword_type : SwordType -> Typetsql)
  nullType = raise internalError
  |
  convertsword_type monoleanType = monolean_typet
  |
  convertsword_type booleanType = boolean_typet
  |
  convertsword_type (stringType(mini,maxi)) = string_typet(mini,maxi)
  |
  convertsword_type (fixedType(p,s)) = fixed_typet(p,s)
  |
  convertsword_type (floatingType(m,e,os)) = floating_typet(m,e,os)
  |
  convertsword_type (enumType(p,t)) = enum_typet(p,table_name t)
  |
  convertsword_type (timeType f) = time_typet f
  |
  convertsword_type (intervalType f) = interval_typet f
  |
  convertsword_type classType = class_typet
  |
  convertsword_type codeType = code_typet
  |
  convertsword_type anyType = any_typet;

```

SML

```

fun (converttype : Typessql -> Typetsql) t = convertsword_type(convertssql_type(t));

```

SML

```

fun (denoteclass_exp : ExpClass -> Valuetsql)
  |
  (variable(v,c)) = v
  |
  denoteclass_exp (constantec c) = denote_classt c;

```

SML

```

fun (lubbound_info : BoundInfo * BoundInfo -> BoundInfo)
  |
  (upb c1,upb c2) = upb(c1 lub c2)
  |
  lubbound_info (constant c1,upb c2) = upb(c1 lub c2)
  |
  lubbound_info (upb c1,constant c2) = upb(c1 lub c2)
  |
  lubbound_info (constant c1,constant c2) = if c1 = c2 then constant c1
  |
  | else upb(c1 lub c2);

```

SML

```

fun (lubexp_class : ExpClass * ExpClass -> ExpClass)
  |
  (variable(v1,c1),constantec c2) =
  |
  | variable(binopt(lub_op,(v1,denote_classt c2)),c1 lub c2)
  |
  lubexp_class (constantec c1,constantec c2) =
  |
  | constantec(c1 lub c2)
  |
  lubexp_class (constantec c1,variable(v2,c2)) =
  |
  | variable(binopt(lub_op,(v2,denote_classt c1)),c1 lub c2)
  |
  lubexp_class (variable(v1,c1),variable(v2,c2)) =
  |
  | variable(binopt(lub_op,(v1,v2)),c1 lub c2);

```

SML

```

fun (lubtype : SwordType * SwordType -> SwordType)(t1,t2) =
  if t1 = t2
  then t1
  else case(t1,t2) of
    (stringType(min1,max1),stringType(min2,max2)) =>
      stringType(min1 min min2,max1 max max2)
  | (fixedType(p1,s1),fixedType(p2,s2)) =>
      fixedType(p1 max p2,s1 max s2)
  | _ => raise wrongType;

```

SML

```

fun (lubcol_type : ColType * ColType -> ColType)
  ((s1,d1),(s2,d2)) = (lubtype(s1,s2),lubtype(d1,d2));

```

The exception *designError* is raised in the following definition. Without the last clause, pattern matching is not exhaustive, although the case of (*dinary*, *dinary*), say, is caught by the *if*.

SML

```

exception designError of string;
fun (lubworth : Worth * Worth -> Worth)(w1,w2) =
  if w1 = w2
  then w1
  else case(w1,w2) of
    (w1,worthless) => w1
  | (worthless,w2) => w2
  | (priceless,w2) => priceless
  | (w1,priceless) => priceless
  | (dinary,sterling) => priceless
  | (sterling,dinary) => priceless
  | _ => raise designError "lubworth";

```

SML

```

fun (lubexp : ExpType * ExpType -> ExpType)
  ((t1,w1),(t2,w2)) = (lubtype(t1,t2),lubworth(w1,w2));

```

SML

```

fun (lubssql_name : SsqlName * SsqlName -> SsqlName)
  (names s1,names s2) = if s1 = s2 then names s1
                        else anons
| lubssql_name (sn1,sn2) = anons;

```

SML

```

fun (lubssql_col : SsqlCol * SsqlCol -> SsqlCol)
  ({name = n1,type_field = t1,col_exist = ce1,col_class = cc1},
   {name = n2,type_field = t2,col_exist = ce2,col_class = cc2})
  =
  {name = lubssql_name(n1,n2),
   type_field = lubcol_type(t1,t2),
   col_exist = ce1 lub ce2,
   col_class = lubbound_info(cc1,cc2)};

```

SML

```

fun (lubtable_info : TableInfo * TableInfo -> TableInfo)
  ({table_exist_class = tec1,table_class = tc1,row_class = rc1},
   {table_exist_class = tec2,table_class = tc2,row_class = rc2})
  =
  {table_exist_class = tec1 lub tec2,
   table_class = tc1 lub tc2,
   row_class = lubbound_info(rc1,rc2)};

```

SML

```

fun (lubtsql_class_name : TsqlClassName * TsqlClassName -> TsqlClassName)
  (nametc s1,nametc s2) = if s1 = s2 then nametc s1
                           else anontc
lubtsql_class_name (constanttc c1,constanttc c2) = if c1 = c2 then constanttc c1
                                                    else anontc
lubtsql_class_name (x,y) = anontc;

```

SML

```

fun (lubtsql_name : TsqlName * TsqlName -> TsqlName)
  (namet s1,namet s2) = if s1 = s2 then namet s1
                        else anont
lubtsql_name (nonet,nonet) = nonet
lubtsql_name (tn1,tn2) = anont;

```

SML

```

fun (lubtsql_col : TsqlCol * TsqlCol -> TsqlCol)
  ({sterling_name = s1,dinary_name= d1,class_name = c1},
   {sterling_name = s2,dinary_name= d2,class_name = c2})
  =
  {sterling_name = lubtsql_name(s1,s2),
   dinary_name = lubtsql_name(d1,d2),
   class_name = lubtsql_class_name(c1,c2)};

```

SML

```

local fun (data_col : TsqlName * TsqlName -> Select_valuetsql list)
      (nonet, nonet) = []
      | data_col (namet s, anont) =
          [anonymous_valuet(contentst(denote_col_spect s))]
      | data_col (namet fs, namet ts) =
          [anonymous_valuet(contentst(denote_col_spect fs))]
      | data_col (nonet, anont) = [anonymous_valuet(denote_nullt)]
      | data_col (anont, nonet) = [anonymous_valuet(denote_nullt)]
      | data_col (n1, n2) = raise internalError
fun (class_col : TsqlClassName * TsqlClassName -> Select_valuetsql list)
      (constanttc fc, constanttc tc) = if fc = tc then []
                                         else raise internalError
      | class_col (nametc f, nametc tn) =
          [anonymous_valuet(contentst(denote_col_spect f))]
      | class_col (constanttc fc, nametc tn) =
          [anonymous_valuet(denote_classt fc)]
      | class_col (nametc f, anontc) =
          [anonymous_valuet(contentst(denote_col_spect f))]
      | class_col (constanttc fc, anontc) =
          [anonymous_valuet(denote_classt fc)]
      | class_col (c1, c2) = raise internalError
in
fun (makesv : TsqlCol * TsqlCol -> Select_valuetsql list)(f, t) =
      data_col(#sterling_name f, #sterling_name t) @
      data_col(#dinary_name f, #dinary_name t) @
      class_col(#class_name f, #class_name t)
end;

```

SML

```

fun (remove_constants : (Col_spectsql, Class)Sum list -> Col_spectsql list) [] = []
| remove_constants ((inL c) :: s) = c :: (remove_constants s)
| remove_constants ((inR c) :: s) = remove_constants s;

```

SML

```

fun (remove_nulls : TsqlRepr list -> TsqlRepr list) [] = []
| remove_nulls (constant_null :: trs) = remove_nulls trs
| remove_nulls (x :: trs) = x :: (remove_nulls trs);

```

SML

```

fun (upper : ExpClass -> Class)
      (variable(c, u)) = u
| upper (constantec u) = u;

```

SML

```

local fun (make_case : Valuetsql * ExpClass -> Valuetsql)
      (data,variable(c,u)) = caset( [data],
                                     [denote_classt(lattice_top())],
                                     c)
    | make_case(data,constantec c) = caset( [data],
                                              [denote_classt(lattice_top())],
                                              denote_classt c)
in
fun (simplifyands : Valuetsql list * ExpClass list -> Valuetsql * ExpClass)
  (vs,cs) =
  let val v = fold(curry binopt and_op)vs
      val c = caset( [v],
                    [fold(curry binopt lub_op)(map denote_class_exp cs)],
                    fold(curry binopt glb_op)(at2 (map make_case) (vs,cs)))
      val u = fold (op lub)(map upper cs)
  in (v,variable(c,u))
  end
end;

```

SML

```

local fun (make_case : Valuetsql * ExpClass -> Valuetsql)
      (data,variable(c,u)) = caset( [data],
                                     [denote_classt(lattice_top())],
                                     c)
    | make_case(data,constantec c) = caset( [data],
                                              [denote_classt(lattice_top())],
                                              denote_classt c)
in
fun (simplifyors : Valuetsql list * ExpClass list -> Valuetsql * ExpClass)
  (vs,cs) =
  let val v = fold(curry binopt and_op)vs
      val c = caset( [v],
                    [fold(curry binopt glb_op)(at2 (map make_case) (vs,cs))],
                    fold(curry binopt lub_op)(map denote_class_exp cs))
      val u = fold (op lub)(map upper cs)
  in (v,variable(c,u))
  end
end;

```

SML

```

fun (constant_value_data : Constant_valuessql -> Valuetsql)
  | denote_nulls = denote_nullt
  | constant_value_data denote_voids = denote_voidt
  | constant_value_data denote_trues = denote_truet
  | constant_value_data denote_falses = denote_falset
  | constant_value_data (denote_strings (s,t)) =
    denote_stringt (s,converttype t)
  | constant_value_data (denote_fixeds (f,t)) =
    denote_fixedt (f,converttype t)
  | constant_value_data (denote_floatings (f,t)) =
    denote_floatingt (f,converttype t)
  | constant_value_data (denote_enums (e,t)) =
    denote_enumt (e,converttype t)
  | constant_value_data (denote_times (tm,t)) =
    denote_timet (tm,converttype t)
  | constant_value_data (denote_intervals (i,t)) =
    denote_intervalt (i,converttype t)
  | constant_value_data (denote_classs c) = denote_classt c
  | constant_value_data (denote_codes c) = denote_codet c;

```

SML

```

local fun (look : Col_specssql -> TsqlRepr)
  cs = lookupcol_spec_dinary(false,convertcol_spec cs)
in
fun (dinary_columns : Col_specssql list -> Col_spectsql list)
  css = map repr_col (remove_nulls(map look css))
end;

```

SML

```

local fun (look : Col_specssql -> TsqlRepr)
  cs = lookupcol_spec_sterling(false,convertcol_spec cs)
in
fun (sterling_columns : Col_specssql list -> Col_spectsql list)
  css = map repr_col (remove_nulls(map look css))
end;

```

SML

```

fun (tuple_list_max_row_class : Tuple_listssql -> ExpClass)
  t = constantec(client_clearance());

```

SML

```

| fun (upb_row_class : TableInfo -> Class)
|     {table_exist_class=tec,table_class=tc,row_class=upb rc} = rc
|     upb_row_class {table_exist_class=tec,table_class=tc,row_class=constant rc} = rc;

```

In the following, the specifications of *internal\_value\_class* and *value\_type* are incomplete. (*all\_binop*, *some\_binop*, *all\_binop\_list* and *some\_binop\_list* are missing from *internal\_value\_class* and *caseVal* is missing from *value\_type*.)

SML

```

| fun (value_data : Valuessql -> Valuetsql)
|     (denote_constants c) = constant_value_data c

```

SML

```

| | value_data (monops (opr,v)) = monopt (opr,value_data v)
| | value_data (binops (opr,(v1,v2))) = binopt (opr,(value_data v1,value_data v2))
| | value_data (triops (opr,(v1,v2,v3))) = triopt (opr,(value_data v1,
| |                                     value_data v2,value_data v3))

```

SML

```

| | value_data (converts (v,t)) = convertt (value_data v,converttype t)
| | value_data (convert_domains (v,domain,t)) =
| |     let val ts = converttable_spec domain
| |         val tn = table_name ts
| |     in convert_domaint (tn,(value_data v,converttype t))
| |     end

```

SML

```

| | value_data (make_sterlings v) = value_data v
| | value_data (make_dinarys v) = value_data v

```

SML

```

| | value_data (declare_s (id,(v1,v2))) =
| | in_new_scope(fn () =>
| |   let val t1 = value_type v1
| |       val data1 = value_data v1
| |       val class1 = value_class v1
| |       val data2 = value_data v2
| |   in case class1 of
| |       variable (c,u) =>
| |         let val (cn,dn) = enter_identifier(id,t1,u)
| |             in declare_t (cn,(c,declare_t(dn,(data1,data2))))
| |         end
| |       | constant_ec c =>
| |         let val dn = enter_identifier_constant_class(id,t1,c)
| |             in declare_t(dn,(data1,data2))
| |         end
| |   end)

```

SML

```

| | value_data (case_val_s (test,caseList,valList,elseVal)) = case_val_t(value_data test,
| |   map value_data caseList,
| |   map value_data valList,
| |   value_data elseVal)
| | value_data (case_s (caseList,valList,elseVal)) = case_t(map value_data caseList,
| |   map value_data valList,
| |   value_data elseVal)

```

SML

```

| | value_data (set_func_all_s (opr,v)) = set_func_all_t (opr,value_data v)
| | value_data (set_func_distinct_s (opr,v)) = set_func_distinct_t (opr,value_data v)
| | value_data (count_non_null_s (v,t)) = count_non_null_t (value_data v,convert_type t)
| | value_data (count_distinct_s (v,t)) = count_distinct_t (value_data v,convert_type t)
| | value_data (count_all_s t) = count_all_t (convert_type t)

```

SML

```

| | value_data (all_binop_s (opr,(v,vs))) = all_binop_t (opr,(value_data v,tuple_list_data vs))
| | value_data (some_binop_s (opr,(v,vs))) =
| |   some_binop_t (opr,(value_data v,tuple_list_data vs))
| | value_data (all_binop_list_s (opr,(v,vs))) =
| |   all_binop_list_t (opr,(value_data v,map value_data vs))
| | value_data (some_binop_list_s (opr,(v,vs))) =
| |   some_binop_list_t (opr,(value_data v,map value_data vs))
| | value_data (exists_tuples_s tuples) = exists_tuples_t (tuple_list_data tuples)
| | value_data (single_value_s v) = single_value_t (tuple_list_data v)

```

SML

```

| | value_data (contents_s cs) =
| |           (case lookup_col_spec_sterling(false, convert_col_spec cs) of
| |             constant_null => denote_null_t
| |             | tr => contents_t(repr_col tr))
| |
| | value_data (sterling_contents_s cs) =
| |           (case lookup_col_spec_sterling(false, convert_col_spec cs) of
| |             constant_null => denote_null_t
| |             | tr => contents_t(repr_col tr))
| |
| | value_data (dinary_contents_s cs) =
| |           (case lookup_col_spec_dinary(false, convert_col_spec cs) of
| |             constant_null => denote_null_t
| |             | tr => contents_t(repr_col tr))

```

SML

```

| | value_data (classification_s col) =
| |           let val cs = class_column col
| |             in if isL cs then (contents_t(getL cs))
| |               else denote_class_t (getR cs)
| |             end

```

SML

```

| | value_data (row_existence_s t) =
| |           let val ts = convert_table_spec t
| |             val rc = lookup_table_row_class(false, ts)
| |             in denote_name rc
| |             end

```

SML

```

| | value_data joined_row_existence_s =
| |           let val ns = lookup_local_row_classes()
| |             in (fold(curry binop_t lub_op)(map denote_name ns))
| |             end

```

SML

```

| | value_data (classify_s(v,c)) = value_data v
| | value_data (classify_default_s v) = value_data v

```

SML

```

| | value_data (observed_s(n,c)) = raise notTrigger
| | value_data (modified_s n) = raise notTrigger

```

SML

```

| value_data (context_s s) = let val (cv,class) = contextual_data s
|                                     in constant_value_data cv
|                                     end
| value_data (parameter_s name) = let val (cv,class) = lookup_param_data name
|                                     in constant_value_data cv
|                                     end

```

SML

```

| and (value_type : Value_ssql -> ExpType)
|     (denote_constant_s c) = (constant_value_type c,worthless)

```

SML

```

| value_type (monop_s (opr,v)) = let val (t,w) = value_type v
|                                     in (monop_type(opr,t),w)
|                                     end

```

SML

```

| value_type (binop_s (opr,(v1,v2))) = let val (t1,w1) = value_type v1
|                                     val (t2,w2) = value_type v2
|                                     in (binop_type (opr,t1,t2),lub_worth(w1,w2))
|                                     end

```

SML

```

| value_type (triop_s (opr,(v1,v2,v3))) =
|     let val (t1,w1) = value_type v1
|         val (t2,w2) = value_type v2
|         val (t3,w3) = value_type v3
|     in (triop_type (opr,t1,t2,t3),lub_worth(w1,lub_worth(w2,w3)))
|     end

```

SML

```

| value_type (convert_s (v,t)) = let val st = convert_ssql_type t
|                                     val (ty,w) = value_type v
|                                     val ok = check_type_conversion_domain(ty,st)
|                                     in (st,w)
|                                     end

```

SML

```

| value_type (convert_domain_s (v,domain,t)) =
|     let val st = convert_ssql_type t
|         val (ty,w) = value_type v
|         val ok = check_type_conversion_domain(ty,st)
|     in (st,w)
|     end

```

SML

```

| |  value_type      (make_sterling_s v) = let val (t,w) = value_type v
| |                                     in (t,sterling)
| |                                     end

```

SML

```

| |  value_type      (make_dinary_s v) = let val (t,w) = value_type v
| |                                     in (t,dinary)
| |                                     end

```

SML

```

| |  value_type      (declare_s (id,(v1,v2))) =
| |                                     let  val et = value_type v1
| |                                     val cl = lattice_top()
| |                                     in
| |                                     in_new_scope(fn () =>
| |                                     let val i = enter_identifier_constant_class(id,et,cl)
| |                                     in value_type v2
| |                                     end)
| |                                     end

```

SML

```

| |  value_type      (case_s (caseList,valList,elseVal)) =
| |                                     let  val ok = map check_boolean(map value_type caseList)
| |                                     in  fold lub_exp(map value_type (valList @ [elseVal]))
| |                                     end

```

SML

```

| |  value_type      (set_func_all_s(opr,v)) = let val (t,w) = value_type v
| |                                     in
| |                                     (set_func_type(opr,t),w)
| |                                     end
| |  value_type      (set_func_distinct_s(opr,v)) = let val (t,w) = value_type v
| |                                     in
| |                                     (set_func_type(opr,t),w)
| |                                     end

```

SML

```

|   value_type      (count_non_null_s(v,t)) =
|                   let   val st = convertssql_type t
|                   val (t,w) = value_type v
|                   in    case st of
|                       fixedType(p,s) => (st,worthless)
|                       |   other => raise wrongType
|                   end
|   value_type      (count_distinct_s(v,t)) =
|                   let   val st = convertssql_type t
|                   val (t,w) = value_type v
|                   in    case st of
|                       fixedType(p,s) => (st,worthless)
|                       |   other => raise wrongType
|                   end

```

SML

```

|   value_type      (count_all_s t) =
|                   let   val st = convertssql_type t
|                   in    case st of
|                       fixedType(p,s) => (st,worthless)
|                       |   other => raise wrongType
|                   end

```

SML

```

|   value_type      (all_binop_s (opr,(v,vs))) =
|   let   val tt = tuple_listtype vs
|   in    if length tt <> 1 then raise tooWide
|         else let   val (t1,w1) = value_type v
|                   val (t2,w2) = hd tt
|                   in (binop_type(opr,t1,t2),lubworth(w1,w2))
|                   end
|   end
|
|   value_type      (some_binop_s (opr,(v,vs))) =
|   let   val tt = tuple_listtype vs
|   in    if length tt <> 1 then raise tooWide
|         else let   val (t1,w1) = value_type v
|                   val (t2,w2) = hd tt
|                   in (binop_type(opr,t1,t2),lubworth(w1,w2))
|                   end
|   end

```

SML

```

| | value_type (all_binop_list_s (opr,(v,vs))) =
| | let fun (binop_exp : Op * ExpType * ExpType -> ExpType)
| | (opr,(t1,w1),(t2,w2)) =
| | (binop_type(opr,t1,t2),lub_worth(w1,w2))
| | val ets2 = map value_type vs
| | val ets1 = seq(length ets2,value_type v)
| | in fold lub_exp (at3(map binop_exp) (seq(length ets2,opr),ets1,ets2))
| | end
| | value_type (some_binop_list_s (opr,(v,vs))) =
| | let fun (binop_exp : Op * ExpType * ExpType -> ExpType)
| | (opr,(t1,w1),(t2,w2)) =
| | (binop_type(opr,t1,t2),lub_worth(w1,w2))
| | val ets2 = map value_type vs
| | val ets1 = seq(length ets2,value_type v)
| | in fold lub_exp (at3(map binop_exp) (seq(length ets2,opr),ets1,ets2))
| | end

```

SML

```

| | value_type (exists_tuples_s tuples) =
| | let val tt = tuple_list_type tuples
| | in (booleanType,worthless)
| | end

```

SML

```

| | value_type (single_value_s tuples) =
| | let val tt = tuple_list_type tuples
| | in if length tt <> 1 then raise tooWide
| | else hd tt
| | end

```

SML

```

| | value_type (contents_s cs) =
| | let val (ti,sc) = lookup_column_info (convert_col_spec cs)
| | in (case (#type_field sc) of
| | (nullType,nullType) => (nullType,sterling)
| | | (st,nullType) => (st,sterling)
| | | (nullType,dt) => (dt,dinary)
| | | (st,dt) => (st,sterling))
| | end

```

SML

```

| |   value_type   (sterling_contents_s cs) =
| |   let         val (ti,sc) = lookup_column_info (convert_col_spec cs)
| |   in         (case (#type_field sc) of
| |               (nullType,nullType) => (nullType,sterling)
| |               | (st,nullType) => (st,sterling)
| |               | (nullType,dt) => raise wrongWorth
| |               | (st,dt) => (st,sterling))
| |   end

```

SML

```

| |   value_type   (dinary_contents_s cs) =
| |   let         val (ti,sc) = lookup_column_info (convert_col_spec cs)
| |   in         (case (#type_field sc) of
| |               (nullType,nullType) => (nullType,dinary)
| |               | (st,nullType) => raise wrongWorth
| |               | (nullType,dt) => (dt,dinary)
| |               | (st,dt) => (dt,dinary))
| |   end

```

SML

```

| |   value_type   (classification_s col) = (classType,worthless)
| |   value_type   (row_existence_s t) = (classType,worthless)
| |   value_type   (joined_row_existence_s) = (classType,worthless)
| |   value_type   (classify_s(v,c)) = (case value_type c of
| |                                     (classType,w) => value_type v
| |                                     | (other,w) => raise wrongType)
| |   value_type   (classify_default_s v) = value_type v
| |   value_type   (observed_s(n,c)) = raise onlyInTriggers
| |   value_type   (modified_s n) = raise onlyInTriggers
| |   value_type   (context_s s) = let val (cv,class) = contextual_data s
| |                               in (constant_value_type cv,worthless)
| |                               end
| |   value_type   (parameter_s name) = let val (cv,class) = lookup_param_data name
| |                                     in (constant_value_type cv,worthless)
| |                                     end

```

SML

```

and (valueclass : Valuessql -> ExpClass)
  v = case internal_valueclass v of
    ands(datas,classes) => let val (v,c) = simplifyands(datas,classes)
                          in c
                          end
    | ors(datas,classes) => let val (v,c) = simplifyors(datas,classes)
                          in c
                          end
    | simple(variable(exp,up)) => variable(exp,up)
    | simple(constantec c) => constantec c

```

SML

```

and (tuple_listdata : Tuple_listssql -> Tuple_listtsql)
  (table_contentss t) = in_new_scope(fn () =>
let   val ts = converttable_spec t
      val tn = converttableSpecification ts
      val (ti,ci,scs,rc,tcs) = gettable_info ts
      fun (sel: TsqlCol -> Select_valuetsql)
          {sterling_name = namet n,
           dinary_name=dn,
           class_name=cn}
          = anonymous_valuet(contentst(denote_col_spect n))
      | sel x = raise internalError
in   all_tuplest(select_valuest(map sel tcs),
               [fromt(table_contentst tn)],denote_truet,[],denote_truet)
end)

```

SML

```

| tuple_listdata (all_tupless(sel,fr,where,gb_sterling,gb_dinary,gb_class,having))
= in_new_scope(fn () =>
let   val fss = map from_specenter fr
      val gs = sterling_columns gb_sterling
      val gd = dinary_columns gb_dinary
      val gc = map class_column gb_class
in   all_tuplest(select_listdata sel,
                fss,valuedata where,gs @ gd @ remove_constants gc,
                valuedata having)
end)

```

SML

```

| | tuple_list_data (distinct_tuples_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having))
| | = in_new_scope(fn () =>
| |   let   val fss = map from_spec_enter fr
| |         val gs = sterling_columns gb_sterling
| |         val gd = dinary_columns gb_dinary
| |         val gc = map class_column gb_class
| |   in distinct_tuples_t(select_list_data sel,
| |                         fss,value_data where,gs @ gd @ remove_constants gc,
| |                         value_data having)
| |   end)

```

SML

```

| | tuple_list_data (evaluate_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having))
| | = in_new_scope(fn () =>
| |   let   val fss = map from_spec_enter fr
| |         val gs = sterling_columns gb_sterling
| |         val gd = dinary_columns gb_dinary
| |         val gc = map class_column gb_class
| |   in evaluate_t(select_list_data sel,
| |                 fss,value_data where,gs @ gd @ remove_constants gc,
| |                 value_data having)
| |   end)

```

SML

```

| | tuple_list_data (tuple_s vals) =
| |   tuple_t(map value_data vals)

```

SML

```

| | tuple_list_data (union_s tls) =
| |   union_t(map tuple_list_data tls)

```

SML

```

| | tuple_list_data (name_columns_s(names,tul))
| |   = tuple_list_data tul

```

SML

```

and (tuple_list_type : Tuple_list_ssql -> ExpType list)
    (table_contents_s ts) =
    let fun (col_info : TableInfo * ConstraintInfo *
           SsqlCol list * TsqlClassName * TsqlCol list
           -> SsqlCol list)
        (ti,ci,scs,rc,tcs) = scs
        fun (t:SsqlCol -> ColType)
            c = #type_field c
        fun (f:ColType -> ExpType)
            (nullType,nullType) = raise internalError
            | f (st,nullType) = (st,sterling)
            | f (nullType,dt) = raise internalError
            | f (st,dt) = (st,sterling)
        in map f(map t(col_info(get_table_info(convert_table_spec ts))))
    end
  
```

SML

```

| tuple_list_type (all_tuples_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having))
  = in_new_scope(fn () =>
    let val fs = map from_spec_enter fr
        val where_ok = check_boolean(value_type where)
        val having_ok = check_boolean(value_type having)
    in select_list_type sel
    end)
  
```

SML

```

| tuple_list_type (distinct_tuples_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having))
  = in_new_scope(fn () =>
    let val fs = map from_spec_enter fr
        val where_ok = check_boolean(value_type where)
        val having_ok = check_boolean(value_type having)
    in select_list_type sel
    end)
  
```

SML

```

| tuple_list_type (evaluate_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having))
  = in_new_scope(fn () =>
    let val fs = map from_spec_enter fr
        val where_ok = check_boolean(value_type where)
        val having_ok = check_boolean(value_type having)
    in select_list_type sel
    end)
  
```

SML

```
|| tuple_list_type (tuple_s vals) = map value_type vals
```

SML

```
|| tuple_list_type (union_s tls) = let val tiss = map tuple_list_type tls
|                                     in   if length tiss = 0
|                                     then raise emptyUnionList
|                                     else map (fold lub_exp)(invert tiss)
|                                     end
```

SML

```
|| tuple_list_type (name_columns_s (names,tul)) = tuple_list_type tul
```

SML

```
and (from_spec_enter : From_spec_ssql -> From_spec_tsql) fs =
let fun (c_names : TsqlClassName -> string list)
      (constant_tc c) = []
|     c_names (name_tc s) = [s]
|     c_names anon_tc = raise internalError
fun (names : TsqlName -> string list)
      none_t = []
|     names (name_t s) = [s]
|     names anon_t = raise internalError
fun (col_names : TsqlCol -> string list) tc
      = names(#sterling_name tc)
      @ names(#dinary_name tc)
      @ c_names(#class_name tc)

in
```

SML

```
(case fs of
  (from_s t) =>
    let val (tn,ti,scs,rc,tcs) = tuple_list_info t
        val tul = tuple_list_make (t,rc,tcs)
        val cor = enter_table(tn,ti,scs,rc,tcs)
        val cns = (case rc of
                    anon_tc => raise internalError
                    | name_tc s => [s]
                    | constant_tc c => [])
                    @ fold (op @) (map col_names tcs)
        in correlate_from_t(cor,name_columns_t(cns,tul))
        end
```

SML

```

|      (correlate_from_s(name,t)) =>
|          let      val (tn,ti,scs,rc,tcs) = tuple_list_info t
|                  val tul = tuple_list_make (t,rc,tcs)
|                  val cor = enter_corr_table(name,tn,ti,scs,rc,tcs)
|                  val cns = (case rc of
|                              anon_tc => raise internalError
|                              |      name_tc s => [s]
|                              |      constant_tc c => [])
|                  @ fold (op @) (map col_names tcs)
|          in      correlate_from_t(cor,name_columns_t(cns,tul))
|          end)
|
|      end

```

SML

```

and (select_list_type : Select_list_ssql -> ExpType list)
|      all_columns_s =
|          let      fun (t : SsqlCol -> ColType) c = #type_field c
|                  fun (f : ColType -> ExpType)
|                      (nullType,nullType) = raise internalError
|                      |      f (st,nullType) = (st,sterling)
|                      |      f (nullType,dt) = (dt,dinary)
|                      |      f (st,dt) = (st,sterling)
|          in      map f(map t(lookup_local_col_info()))
|          end
|
|      select_list_type (select_values_s vals) = map select_value_type vals

```

SML

```

and (select_value_type : Select_value_ssql -> ExpType)
|      (anonymous_value_s v) = value_type v
|
|      select_value_type (named_value_s(name,v)) = value_type v
|
|      select_value_type (anonymous_pair_s(sval,dval)) = value_type sval
|
|      select_value_type (named_pair_s(name,(sval,dval))) = value_type sval

```

SML

```

and (tuple_list_info : Tuple_list_ssql -> TableName * TableInfo * SsqlCol list
|                                     * TsqlClassName * TsqlCol list)
|
|      (table_contents_s t) =
|          let      val ts = convert_table_spec t
|                  val (ti,ci,scs,rc,tcs) = get_table_info ts
|          in      (name_tn ts,ti,scs,rc,tcs)
|          end

```

SML

```

| tuple_list_info (all_tuples_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having)) =
|   let   val (tis,scss,rcs,tcss) = split4 (map from_spec_info fr)
|   in   in_new_scope(fn () =>
|         let   val fs = map from_spec_enter fr
|               val sli = select_list_info sel
|         in   (anon_tn,fold lub_table_info tis,sli,
|                 fold lub_tsql_class_name rcs,map_col_target sli)
|         end)
|   end

```

SML

```

| tuple_list_info (distinct_tuples_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having)) =
|   let   val (tis,scss,rcs,tcss) = split4 (map from_spec_info fr)
|   in   in_new_scope(fn () =>
|         let   val fs = map from_spec_enter fr
|               val sli = select_list_info sel
|         in   (anon_tn,fold lub_table_info tis,sli,
|                 fold lub_tsql_class_name rcs,map_col_target sli)
|         end)
|   end

```

SML

```

| tuple_list_info (evaluate_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having)) =
|   let   val (tis,scss,rcs,tcss) = split4 (map from_spec_info fr)
|   in   in_new_scope(fn () =>
|         let   val fs = map from_spec_enter fr
|               val sli = select_list_info sel
|         in   (anon_tn,fold lub_table_info tis,sli,
|                 fold lub_tsql_class_name rcs,map_col_target sli)
|         end)
|   end

```

SML

```

| tuple_list_info (tuple_s vals) =
|   let   val   ti = {table_exist_class = query_constants_class(),
|                   table_class = query_constants_class(),
|                   row_class = constant(query_constants_class())}
|         val scs = map value_info vals
|         val tcs = map col_target scs
|   in   (anon_tn,ti,scs,constant_tc(query_constants_class()),tcs)
|   end

```

SML

```

| tuple_list_info (union_s tls) =
|   if length tls = 1
|   then tuple_list_info(hd tls)
|   else let val (tts,tis,scss,rsc,tcss) = split5 (map tuple_list_info tls)
|          val ti = fold lub_table_info tis
|          val scs = map(fold lub_ssql_col)(invert scss)
|          val rc = fold lub_tsql_class_name rsc
|          val tcs = map(fold lub_tsql_col) (invert tcss)
|          in (anon_tn,ti,scs,rc,tcs)
|          end

```

SML

```

| tuple_list_info (name_columns_s (names,tul)) =
|   let fun (merge : string * SsqlCol -> SsqlCol)
|        (n,{name=cn,type_field=ty,col_exist=ce,col_class=cc})
|        = {name=name_s n,type_field=ty,col_exist=ce,col_class=cc}
|   val (tn,ti,scs,rc,tcs) = tuple_list_info tul
|   in (tn,ti,at2(map merge)(names,scs),rc,tcs)
|   end

```

SML

```

and (tuple_list_make : Tuple_listssql * TsqlClassName * TsqlCol list -> Tuple_listtsql)
  (table_contentss t,to_rc,to_tcs) =
  let
    val (ti,ci,scs,rc,tcs) = gettable_info (converttable_spec t)
    val tc = table_contentst(converttableSpecification(converttable_spec t))
    fun (mk : TsqlName * TsqlName -> Valuetsql list)
        (namet f,namet tn) = [contentst(denote_col_spect f)]
        | mk (namet f,anont) = [contentst(denote_col_spect f)]
        | mk (namet f,nonet) = raise internalError
        | mk (anont,namet tn) = raise internalError
        | mk (anont,anont) = raise internalError
        | mk (anont,nonet) = raise internalError
        | mk (nonet,namet tn) = [denote_nullt]
        | mk (nonet,anont) = [denote_nullt]
        | mk (nonet,nonet) = []
    fun (mkc : TsqlClassName * TsqlClassName -> Valuetsql list)
        (nametc f,nametc t) = [contentst(denote_col_spect f)]
        | mkc (nametc f,constanttc c) = raise internalError
        | mkc (constanttc c,nametc tn) = [denote_classt c]
        | mkc (constanttc c1,constanttc c2) =
            if c1 = c2 then []
            else raise internalError
        | mkc (anontc,x) = raise internalError
        | mkc (nametc f,anontc) = [contentst(denote_col_spect f)]
        | mkc (constanttc c,anontc) = [denote_classt c]
    fun (col : TsqlCol * TsqlCol -> Valuetsql list)
        ({sterling_name= from_sn,dinary_name = from_dn,
          class_name = from_cn},
         {sterling_name= to_sn,dinary_name = to_dn,
          class_name = to_cn}) =
        mk(from_sn,to_sn) @ mk(from_dn,to_dn)
        @ mkc(from_cn,to_cn)
  in
    if to_rc = rc andalso to_tcs = tcs
    then tc
    else let val vals = mkc(rc,to_rc) @
        (fold (op @)(at2(map col)(tcs,to_tcs)))
        in all_tuplest(select_valuest(map anonymous_valuet vals),
          [fromt tc],denote_truet,[],denote_truet)
        end
  end
end

```

SML

```

tuple_list_make (all_tuples_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having)
                  ,to_rc,to_tcs)
= in_new_scope(fn () =>
  let    val fs = map from_spec_enter fr
         val gbs = sterling_columns gb_sterling
         val gbd = dinary_columns gb_dinary
         val gbc = remove_constants(map class_column gb_class)
  in all_tuples_t(select_values_t(select_list_make(sel,to_rc,to_tcs)),
                  fs,value_data where,gbs @ gbd @ gbc,
                  value_data having)
  end)

```

SML

```

tuple_list_make (distinct_tuples_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having)
                  ,to_rc,to_tcs)
= in_new_scope(fn () =>
  let    val fs = map from_spec_enter fr
         val gbs = sterling_columns gb_sterling
         val gbd = dinary_columns gb_dinary
         val gbc = remove_constants(map class_column gb_class)
  in distinct_tuples_t(select_values_t(select_list_make(sel,to_rc,to_tcs)),
                       fs,value_data where,gbs @ gbd @ gbc,
                       value_data having)
  end)

```

SML

```

tuple_list_make (evaluate_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having)
                  ,to_rc,to_tcs)
= in_new_scope(fn () =>
  let    val fs = map from_spec_enter fr
         val gbs = sterling_columns gb_sterling
         val gbd = dinary_columns gb_dinary
         val gbc = remove_constants(map class_column gb_class)
  in evaluate_t(select_values_t(select_list_make(sel,to_rc,to_tcs)),
                fs,value_data where,gbs @ gbd @ gbc,
                value_data having)
  end)

```

SML

```

| tuple_list_make (tuple_s vals,to_rc,to_tcs) =
|   let    val tvals = (case to_rc of
|                 constant_tc c => []
|                 other => [(denote_class_t (query_constants_class()))])
|         @ (fold(op @)(at2(map make_col)(vals,to_tcs)))
|   in tuple_t tvals
|   end

```

SML

```

| tuple_list_make (union_s tls,to_rc,to_tcs) =
|   if length tls <> 1
|   then tuple_list_make(hd tls,to_rc,to_tcs)
|   else union_t(at3(map tuple_list_make)
|               (tls,seq(length tls,to_rc),seq(length tls,to_tcs)))

```

SML

```

| tuple_list_make (name_columns_s(names,tul),to_rc,to_tcs)
|   = tuple_list_make(tul,to_rc,to_tcs)

```

SML

```

and (from_spec_info : From_spec_ssql ->
|   TableInfo * SsqlCol list * TsqlClassName * TsqlCol list)
|   (from_s t) = let    val (tn,ti,scs,rc,tcs) = tuple_list_info t
|                 in    (ti,scs,rc,tcs)
|                 end
|   from_spec_info (correlate_from_s(cn,t)) = let    val (tn,ti,scs,rc,tcs) = tuple_list_info t
|                 in    (ti,scs,rc,tcs)
|                 end

```

SML

```

and (select_list_info : Select_list_ssql -> SsqlCol list)
|   all_columns_s = lookup_local_col_info()
|   select_list_info (select_values_s vals) = map select_value_info vals

```

SML

```

and (select_value_info : Select_value_ssql -> SsqlCol)
|   (anonymous_value_s v) = value_info v

```

SML

```

| select_value_info(named_value_s(n,v)) =
|   let    val {name = nm,type_field = ty,col_exist = ce,col_class = cc}
|         = value_info v
|   in    {name = name_s n,type_field = ty,col_exist = ce,col_class = cc}
|   end

```

SML

```

| |   select_value_info(anonymous_pair_s(sval,dval)) =
| |       let   val (stype,sw) = value_type sval
| |             val (dtype,dw) = value_type dval
| |             val u = upper(value_class sval)
| |                   lub upper(value_class dval)
| |       in   {name = anon_s,
| |             type_field = (stype,dtype),
| |             col_exist = query_constants_class(),
| |             col_class = upb u}
| |       end

```

SML

```

| |   select_value_info(named_pair_s(n,(sval,dval))) =
| |       let   val (stype,sw) = value_type sval
| |             val (dtype,dw) = value_type dval
| |             val u = upper(value_class sval)
| |                   lub upper(value_class dval)
| |       in   {name = name_s n,
| |             type_field = (stype,dtype),
| |             col_exist = query_constants_class(),
| |             col_class = upb u}
| |       end

```

SML

```

| and   (value_info : Value_ssql -> SsqlCol)
|       v = case value_class v of
|           variable(exp,c) => {name= anon_s,
|                               type_field= col_exp(value_type v),
|                               col_exist=query_constants_class(),
|                               col_class=upb c}
|       |   constant_ec c => {name= anon_s,
|                               type_field= col_exp(value_type v),
|                               col_exist=query_constants_class(),
|                               col_class=constant c}

```

SML

```

| and   (internal_value_class : Value_ssql -> InternalExpClass)
|       (denote_constant_s c) =
|           simple(constant_ec(query_constants_class()))
| |   internal_value_class (monop_s(opr,v)) = simple(value_class v)

```

SML

```

|   internal_value_class (binops(or_op,(val1,val2))) =
|   let
|     val (v1,c1) =
|       (case (internal_value_class val1) of
|         ands(datas,classes) =>
|           split[simplify_ands(datas,classes)]
|         | ors(datas,classes) => (datas,classes)
|         | simple e => ([value_data val1],[e]))
|     val (v2,c2) =
|       (case (internal_value_class val2) of
|         ands(datas,classes) =>
|           split[simplify_ands(datas,classes)]
|         | ors(datas,classes) => (datas,classes)
|         | simple e => ([value_data val2],[e]))
|   in ors(v1 @ v2,c1 @ c2)
|   end

```

SML

```

|   internal_value_class (binops(and_op,(val1,val2))) =
|   let
|     val (v1,c1) =
|       (case (internal_value_class val1) of
|         ands(datas,classes) => (datas,classes)
|         | ors(datas,classes) =>
|           split[simplify_ors(datas,classes)]
|
|         | simple e => ([value_data val1],[e]))
|     val (v2,c2) =
|       (case (internal_value_class val2) of
|         ands(datas,classes) => (datas,classes)
|         | ors(datas,classes) =>
|           split[simplify_ors(datas,classes)]
|
|         | simple e => ([value_data val2],[e]))
|   in ors(v1 @ v2,c1 @ c2)
|   end

```

SML

```

| |   internal_value_class (binop_s(opr,(val1,val2))) =
| |       let   val vc1 = value_class val1
| |           val vc2 = value_class val2
| |       in   simple(lub_exp_class(vc1,vc2))
| |       end

```

SML

```

| |   internal_value_class (triop_s(opr,(val1,val2,val3))) =
| |       let   val vc1 = value_class val1
| |           val vc2 = value_class val2
| |           val vc3 = value_class val3
| |       in   simple(lub_exp_class(vc1,lub_exp_class(vc2,vc3)))
| |       end

```

SML

```

| |   internal_value_class (convert_s(v,t)) = simple(value_class v)
| |   internal_value_class (convert_domain_s(v,domain,t)) = simple(value_class v)
| |   internal_value_class (make_sterling_s v) = simple(value_class v)
| |   internal_value_class (make_dinary_s v) = simple(value_class v)

```

SML

```

| |   internal_value_class (declare_s (id,(val1,val2))) =
| |       let
| |           val t1 = value_type val1
| |           val data1 = value_data val1
| |           val class1 = value_class val1
| |           val class2 = value_class val2
| |           val class2exp = denote_class_exp class2
| |       in
| |           in_new_scope(fn () =>
| |               let val c =
| |                   case class1 of
| |                       variable (c,u) =>
| |                           let val (cn,dn) = enter_identifier(id,t1,u)
| |                               in   declare_t (cn,(c,
| |                                       declare_t(dn,(data1,class2exp))))
| |                           end
| |                       |
| |                       constant_ec c =>
| |                           let val dn = enter_identifier_constant_class(id,t1,c)
| |                               in declare_t(dn,(data1,class2exp))
| |                           end
| |                   in   simple(variable(c,upper class2))
| |               end)
| |       end
| |   end

```

SML

```

|   internal_value_class (case Val_s (test,caseList,valList,elseVal)) =
|     let   val tcn = unique_name()
|           val tc = contents_t(denote_col_spec_t tcn)
|           val tvn = unique_name()
|           val tv = contents_t(denote_col_spec_t tvn)
|           fun (limb1 : Value_ssql * Value_ssql
|               -> Value_tsql list * Value_tsql list)
|               (e,v) =
|                 let   val v1 = denote_class_exp(value_class e)
|                       val t1 = monopt(not_op,binopt(dom_op,
|               (denote_class_t(query_class()),v1)))
|                       val t2 = binopt(equal_op,(tv,value_data e))
|                       val v2 = denote_class_exp(value_class v)
|                 in   ([t1,t2],[v1,v2])
|                 end
|           fun (limb : Value_ssql list * Value_ssql list
|               -> Value_tsql list * Value_tsql list)
|               (es,vs) =
|                 let val (ess,vss) = split(at2(map limb1)(es,vs))
|                 in (fold(op @)ess,fold(op @)vss)
|                 end
|           fun (uppers : ExpClass list -> Class)
|               es = fold (op lub) (map upper es)
|           val (c1,v1) = limb(caseList,valList)
|           val check_list = case_t(c1,v1,
|               denote_class_exp(value_class elseVal))
|           val check_test = case_t([binopt(dom_op,(denote_class_t
|               (query_class()),tc))],
|               [check_list],
|               tc)
|           val c = declare_t(tcn,(denote_class_exp(value_class test),
|               declare_t(tvn,(value_data test,check_test))))
|           val u = uppers(map value_class
|               (caseList @ valList @ [test,elseVal]))
|   in simple(variable(c,u))
|   end

```

SML

```

|   internal_value_class (case_s (caseList, valList, elseVal)) =
|       let fun (limb1 : Value_ssql * Value_ssql
|               -> Value_tsql list * Value_tsql list)
|           (e,v) =
|               let val v1 = denote_class_exp(value_class e)
|                   val t1 = monop_t(not_op, binop_t(dom_op,
|               (denote_class_t(query_class()), v1)))
|                   val v2 = denote_class_exp(value_class v)
|               in ([t1, value_data e], [v1, v2])
|               end
|           fun (limb : Value_ssql list * Value_ssql list
|               -> Value_tsql list * Value_tsql list)
|           (es, vs) =
|               let val (ess, vss) = split(at2(map limb1)(es, vs))
|               in (fold(op @) ess, fold(op @) vss)
|               end
|           fun (uppers : ExpClass list -> Class)
|               es = fold (op lub) (map upper es)
|               val (c1, v1) = limb(caseList, valList)
|               val c = case_t(c1, v1, denote_class_exp(value_class elseVal))
|               val u = uppers(map value_class
|                   (caseList @ valList @ [elseVal]))
|           in simple(variable(c, u))
|           end

```

SML

```

|   internal_value_class (set_func_all_s (and_op, v)) =
|       (case value_class v of
|         variable(c, u) =>
|           let val cf = case_t([value_data v],
|                               [denote_class_t(lattice_top()),
|                               c])
|               val ce = case_t([set_func_all_t(and_op, value_data v)],
|                               [set_func_all_t(lub_op, c)],
|                               set_func_all_t(glb_op, cf))
|           in simple(variable(ce, u))
|           end
|         constant_ec c => simple(constant_ec c))

```

SML

```

|   internal_value_class (set_func_all_s (or_op,v)) =
|   (case value_class v of
|     variable(c,u) =>
|       let    val ct = case_t([value_data v],
|                             [c],
|                             denote_class_t(lattice_top()))
|           val ce = case_t([set_func_all_t(or_op,value_data v)],
|                           [set_func_all_t(glb_op,ct)],
|                           set_func_all_t(lub_op,c))
|       in    simple(variable(ce,u))
|       end
|   |   constant_ec c => simple(constant_ec c))

```

SML

```

|   internal_value_class (set_func_all_s (opr,v)) =
|   (case value_class v of
|     variable(c,u) =>
|       simple(variable(set_func_all_t(lub_op,c),u))
|   |   constant_ec c => simple(constant_ec c))

```

SML

```

|   internal_value_class (set_func_distinct_s (and_op,v)) =
|   (case value_class v of
|     variable(c,u) =>
|       let    val cf = case_t([value_data v],
|                             [denote_class_t(lattice_top())],
|                             c)
|           val ce = case_t([set_func_all_t(and_op,value_data v)],
|                           [set_func_all_t(lub_op,c)],
|                           set_func_all_t(glb_op,cf))
|       in    simple(variable(ce,u))
|       end
|   |   constant_ec c => simple(constant_ec c))

```

SML

```

| |   internal_value_class (set_func_distinct_s (or_op,v)) =
| |   (case value_class v of
| |     variable(c,u) =>
| |       let    val ct = case_t([value_data v],
| |                             [c],
| |                             denote_class_t(lattice_top()))
| |           val ce = case_t([set_func_all_t(or_op,value_data v)],
| |                           [set_func_all_t(glb_op,ct)],
| |                           set_func_all_t(lub_op,c))
| |       in    simple(variable(ce,u))
| |       end
| |   |   constant_ec c => simple(constant_ec c))

```

SML

```

| |   internal_value_class (set_func_distinct_s (opr,v)) =
| |   (case value_class v of
| |     variable(c,u) =>
| |       simple(variable(set_func_all_t(lub_op,c),u))
| |   |   constant_ec c => simple(constant_ec c))

```

SML

```

| |   internal_value_class (count_non_null_s (v,t)) =
| |   (case value_class v of
| |     variable(c,u) =>
| |       simple(variable(set_func_all_t(lub_op,c),u))
| |   |   constant_ec c => simple(constant_ec c))

```

SML

```

| |   internal_value_class (count_distinct_s (v,t)) =
| |   (case value_class v of
| |     variable(c,u) =>
| |       simple(variable(set_func_all_t(lub_op,c),u))
| |   |   constant_ec c => simple(constant_ec c))

```

SML

```

| |   internal_value_class (count_all_s t) =
      let val ti = fold lub_table_info(lookup_local_table_info())
      in   case #row_class ti of
          upb u =>
            let   val c = fold(curry binop_t lub_op)(map
                          denote_name(lookup_local_row_classes()))
            in simple(variable(c,u))
            end
          |   constant u => simple(constant_ec u)
      end
  
```

SML

```

| |   internal_value_class (exists_tuples_s tuples) =
      let val (tn,ti,scs,rc,tcs) = tuple_list_info tuples
      in   case #row_class ti of
          upb u =>
            let val c = denote_class_exp
                          (tuple_list_max_row_class tuples)
            in simple(variable(c,u))
            end
          |   constant u => simple(constant_ec u)
      end
  
```

SML

```

| |   internal_value_class (single_value_s v) =
      let val (tn,ti,scs,rc,tcs) = tuple_list_info v
      in   case #col_class (hd scs) of
          upb u =>
            let val c = single_value_t (tuple_list_class v)
            in simple(variable(c,u))
            end
          |   constant u => simple(constant_ec u)
      end
  
```

SML

```

| |   internal_value_class (contents_s col) =
      let val (tr,u) = lookup_col_spec_class(false,convert_col_spec col)
      in   case tr of
          constant_class c => simple(constant_ec c)
          |   other => simple(variable(denote_name tr,u))
      end
  
```

SML

```

| |   internal_value_class (sterling_contents_s col) =
| |   let val (tr,u) = lookup_col_spec_class(false,convert_col_spec col)
| |   in   case tr of
| |       constant_class c => simple(constant_ec c)
| |       |   other => simple(variable(denote_name tr,u))
| |   end

```

SML

```

| |   internal_value_class (dinary_contents_s col) =
| |   let val (tr,u) = lookup_col_spec_class(false,convert_col_spec col)
| |   in   case tr of
| |       constant_class c => simple(constant_ec c)
| |       |   other => simple(variable(denote_name tr,u))
| |   end

```

SML

```

| |   internal_value_class (classification_s col) =
| |   let val (ti,ci) = lookup_column_info(convert_col_spec col)
| |   in   case #row_class ti of
| |       upb u =>
| |         let   val rc = lookup_column_row_class
| |               (false,convert_col_spec col)
| |               val c = denote_name rc
| |               in   simple(variable(c,u))
| |               end
| |       |   constant c => simple(constant_ec c)
| |   end

```

SML

```

| |   internal_value_class (row_existence_s t) =
| |   let val td = lookup_table_detail(convert_table_spec t)
| |   in   simple(constant_ec (#table_class(#info td)))
| |   end

```

SML

```

| |   internal_value_class joined_row_existence_s =
| |   let val ti = fold_lub_table_info(lookup_local_table_info())
| |   in   simple(constant_ec (#table_class ti))
| |   end

```

SML

```

| |   internal_value_class (classifys(v,c)) = simple(valueclass c)
| |   internal_value_class (classifys-defaults v) = simple(constantec(query_class()))

```

SML

```

| |   internal_value_class (observeds(n,c)) = raise notTrigger
| |   internal_value_class (modifieds n) = raise notTrigger
| |   internal_value_class (contexts t) =
| |       let val (cv,class) = contextual_data t
| |       in simple(constantec class)
| |       end
| |   internal_value_class (parameters name) =
| |       let val (cv,class) = lookupparam_data name
| |       in simple(constantec class)
| |       end

```

SML

```

and (tuple_listclass : Tuple_listssql -> Tuple_listtsql)
    (table_contentss t) = in_new_scope(fn () =>
    let   val ts = converttable_spec t
        val tn = converttableSpecification ts
        val (ti,ci,scs,rc,tcs) = gettable_info ts
        fun (sel: TsqlCol -> Select_valuetsql)
            {sterling_name = sn,
             dinary_name=dn,
             class_name=nametc n}
            = anonymous_valuet(contentst(denote_col_spect n))
        | sel {sterling_name = sn,
              dinary_name=dn,
              class_name=constanttc c}
            = anonymous_valuet(denote_classt c)
        | sel x = raise internalError
    in   all_tuplest(select_valuest(map sel tcs),
                    [fromt(table_contentst tn)],denote_truet,[],denote_truet)
    end)

```

SML

```

| | tuple_list_class (all_tuples_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having))
| | = in_new_scope(fn () =>
| |   let   val fss = map from_spec_enter fr
| |         val gs = sterling_columns gb_sterling
| |         val gd = dinary_columns gb_dinary
| |         val gc = map class_column gb_class
| |   in all_tuples_t(select_list_class sel,
| |                   fss,value_data where,gs @ gd @ remove_constants gc,
| |                   value_data having)
| |   end)

```

SML

```

| | tuple_list_class (distinct_tuples_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having))
| | = in_new_scope(fn () =>
| |   let   val fss = map from_spec_enter fr
| |         val gs = sterling_columns gb_sterling
| |         val gd = dinary_columns gb_dinary
| |         val gc = map class_column gb_class
| |   in distinct_tuples_t(select_list_class sel,
| |                        fss,value_data where,gs @ gd @ remove_constants gc,
| |                        value_data having)
| |   end)

```

SML

```

| | tuple_list_class (evaluate_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having))
| | = in_new_scope(fn () =>
| |   let   val fss = map from_spec_enter fr
| |         val gs = sterling_columns gb_sterling
| |         val gd = dinary_columns gb_dinary
| |         val gc = map class_column gb_class
| |   in evaluate_t(select_list_class sel,
| |                 fss,value_data where,gs @ gd @ remove_constants gc,
| |                 value_data having)
| |   end)

```

SML

```

| | tuple_list_class (tuple_s vals) =
| |   tuple_t(map denote_class_exp(map value_class vals))

```

SML

```

| | tuple_list_class (union_s tls) =
| |   union_t(map tuple_list_class tls)

```

SML

```

| tuple_list_class (name_columns_s (names,tul))
|                                     = tuple_list_class tul
and (select_list_class : Select_list_ssql -> Select_list_tsql)
    all_columns_s =
        let val (trs,classes) = split(lookup_local_col_spec_classes())
        in select_values_t(map anonymous_value_t (map denote_name trs))
        end
| select_list_class (select_values_s vals) = select_values_t(map select_value_class vals)

```

SML

```

and (select_value_class : Select_value_ssql -> Select_value_tsql)
    (anonymous_value_s v) =
        anonymous_value_t(denote_class_exp(value_class v))
| select_value_class (named_value_s(name,v)) =
        anonymous_value_t(denote_class_exp(value_class v))
| select_value_class (anonymous_pair_s(sval,dval)) =
        anonymous_value_t(denote_class_exp(value_class sval))
| select_value_class (named_pair_s(name,(sval,dval))) =
        anonymous_value_t(denote_class_exp(value_class sval))

```

SML

SML

```

and (select_list_make : Select_list_ssql * TsqlClassName * TsqlCol list -> Select_value_tsql list)
    (all_columns_s,to_rc,to_tcs) =
        let    val rcs = lookup_local_row_classes()
        val rc = (fold(curry binop_t lub_op)(map denote_name rcs))
        in
        (case to_rc of
            anon_tc => [anonymous_value_t rc]
          | name_tc s => [anonymous_value_t rc]
          | constant_tc c => [])
        end
        @
        (fold (op @)(at2(map make_sv)
            (lookup_local_col_implementation(),to_tcs)))

```

SML

```

|   select_listmake (select_valuess vals,to_rc,to_tcs) =
|       let fun (make : TsqlRepr -> Valuetsql )
|           (local_identifier s) = raise internalError
|           | make (column(t,c)) = contentst(absolute_col_spect([],t,c))
|           | make (constant_class c) = denote_classt c
|           | make constant_null = denote_nullt
|       in (case to_rc of
|           anontc =>
|           let val rcvs = map make(lookuplocal_row_classes())
|           in [anonymous_valuet
|               (fold(curry binopt lub_op)rcvs)]
|           end
|           | nametc s =>
|           let val rcvs = map make(lookuplocal_row_classes())
|           in [anonymous_valuet
|               (fold(curry binopt lub_op)rcvs)]
|           end
|           | constanttc c => []
|       @
|       (fold (op @)(at2(map select_valuemake)(vals,to_tcs)))
|   end

```

SML

```

| and (select_valuemake : Select_valuessql * TsqlCol -> Select_valuetsql list)
|   (anonymous_values v,tcol) =
|       map anonymous_valuet(make_col(v,tcol))
|   select_valuemake (named_values(name,v),tcol) =
|       map anonymous_valuet(make_col(v,tcol))
|   select_valuemake (anonymous_pairs(sval,dval),tcol) =
|       let val sv = anonymous_valuet(valuedata sval)
|           val sd = anonymous_valuet(valuedata dval)
|       in [sv,sd]
|       end
|   select_valuemake (named_pairs(name,(sval,dval)),tcol) =
|       let val sv = anonymous_valuet(valuedata sval)
|           val sd = anonymous_valuet(valuedata dval)
|       in [sv,sd]
|       end

```

SML

```

and (make_col : Valuessql * TsqlCol -> Valuetsql list)
    (v,tcol) = let val et = valuetype v
              in
                (case (#sterling_name tcol) of
                  nonet => []
                | anont => [makesterling(v,et)]
                | namet s => [makesterling(v,et)])
                @
                (case (#dinary_name tcol) of
                  nonet => []
                | anont => [makedinary(v,et)]
                | namet s => [makedinary(v,et)])
                @
                (case (#class_name tcol) of
                  anontc => [denoteclass_exp(valueclass v)]
                | nametc s => [denoteclass_exp(valueclass v)]
                | constanttc c => [])
              end

```

SML

```

and (makedinary : Valuessql * ExpType -> Valuetsql)
    (v,(t,priceless)) = raise wrongType
| makedinary (v,(t,sterling)) = denotenullt
| makedinary (v,(t,dinary)) = valuedata v
| makedinary (v,(t,worthless)) = denotenullt

```

SML

```

and (makesterling : Valuessql * ExpType -> Valuetsql)
    (v,(t,priceless)) = raise wrongType
| makesterling (v,(t,sterling)) = valuedata v
| makesterling (v,(t,dinary)) = denotenullt
| makesterling (v,(t,worthless)) = valuedata v

```

SML

```

and (select_listdata : Select_listssql -> Select_listtsql)
    all_columnss = select_valuest(map anonymous_valuet (map contentst
                                                    (all_data_columnslocal()))))
| select_listdata (select_valuess vals) = select_valuest(map select_valuedata vals)

```

SML

```

and (select_value_data : Select_valuessql -> Select_valuetsql)
    (anonymous_values v) = anonymous_valuet(value_data v)
| select_value_data (named_values(name,v)) = anonymous_valuet(value_data v)
| select_value_data (anonymous_pairs(sval,dval)) = anonymous_valuet(value_data sval)
| select_value_data (named_pairs(name,(sval,dval))) =
    anonymous_valuet(value_data sval)

```

SML

```

fun (tuple_list_make_outer : Tuple_listssql * bool * TsqlClassName * TsqlCol list
    -> Tuple_listtsql * Querytsql list )
    (table_contentss t,scw,to_rc,to_tcs) =
let    val ts = converttable_spec t
    val (ti,ci,scs,rc,tcs) = gettable_info ts
    val where_class = denote_classt(query_constants_class())
    val sl = (case scw of
        true => [anonymous_valuet where_class]
        | false => [])
    @
    (case to_rc of
        constanttc c => []
        | other =>
            (case rc of
                anontc => raise internalError
                | nametc col =>
                    [anonymous_valuet(contentst
                        (denote_col_spect col))]
                | constanttc cl =>
                    [anonymous_valuet(denote_classt cl)]))
    @ (fold (op @) (at2(map makesv)(tcs,to_tcs)))
    val fs = fromt(table_contentst(converttableSpecification ts))
    val sel_q = all_tuplest(select_valuest sl,[fs],denote_truet,
        [],denote_truet)
in    (sel_q,[])
end

```

SML

```

tuple_list_make_outer (all_tuples_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having)
                      ,scw,to_rc,to_tcs)
= in_new_scope(fn () =>
let   val fs = map from_spec_enter fr
      val sel_vals = select_list_make(sel,to_rc,to_tcs)
      val where_v = value_data where
      val where_c = value_class where
      val where_class = case where_c of
                          variable(exp,c) => exp
                          |   constant_ec c => denote_class_t c
      val sel_list = case scw of
                      true => [anonymous_value_t where_class]@ sel_vals
                      | false => sel_vals
      val sl = select_values_t sel_list
      val cc = denote_class_t(client_clearance())
      val wchk = case where_c of
                  variable(exp,c) =>
                    if client_clearance() dom c
                    then where_v
                    else let val ok = binop_t(dom_op,(cc,exp))
                        in binop_t(or_op,(where_v,
                                      monop_t(not_op,ok)))
                        end
                  |   constant_ec c =>
                    if client_clearance() dom c
                    then where_v
                    else denote_true_t
      val max_rc = fold (op lub)(map upb_row_class
                                   (lookup_local_table_info()))
      val rcs = map denote_name(lookup_local_row_classes())
      val rvis = binop_t(dom_op,(cc,fold(curry binop_t lub_op)rcs))
      val w = if client_clearance() dom max_rc
              then wchk
              else binop_t(and_op,(wchk,rvis))
      val gbs = sterling_columns gb_sterling
      val gbd = dinary_columns gb_dinary
      val gbc = remove_constants(map class_column gb_class)

```

SML

```

val groupby_test = fold(op @)(map column_data_test
                             (gb_sterling @ gb_dinary))
val having_test = case value_class having of
                   constantec c =>
                     if client_clearance() dom c
                     then []
                     else [denote_falset]
                   | variable(exp,c) =>
                     if client_clearance() dom c
                     then []
                     else [binopt(dom_op,(cc,exp))]
val tests = groupby_test @ having_test
val checks =
  if length tests = 0
  then []
  else
    let val t = monopt(not_op,
                      (fold(curry binopt and_op)tests))
        val bad = if client_clearance() dom max_rc
                  then [where_v,t]
                  else [rvis,where_v,t]
        val junk = anonymous_valuet(denote_truet)
        val junk_sl = select_valuest[junk]
    in [selectt(all_tuplest(junk_sl,fs,
                          fold(curry binopt and_op)bad,[],denote_truet))]
    end
val sel_q = all_tuplest(sl,fs,w,gb @ gbd @ gbc,value_data having)
in (sel_q,checks)
end)

```

SML

```

tuple_list_make_outer (distinct_tuples_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having)
                      ,scw,to_rc,to_tcs)
= in_new_scope(fn () =>
let   val fs = map from_spec_enter fr
      val sel_vals = select_list_make(sel,to_rc,to_tcs)
      val where_v = value_data where
      val where_c = value_class where
      val where_class = case where_c of
                          variable(exp,c) => exp
                          | constant_ec c => denote_class_t c
      val sel_list = case scw of
                      true => [anonymous_value_t where_class]@ sel_vals
                      | false => sel_vals
      val sl = select_values_t sel_list
      val cc = denote_class_t(client_clearance())
      val wchk = case where_c of
                  variable(exp,c) =>
                    if client_clearance() dom c
                    then where_v
                    else let val ok = binop_t(dom_op,(cc,exp))
                        in binop_t(or_op,(where_v,
                                    monop_t(not_op,ok)))
                        end
                  | constant_ec c =>
                    if client_clearance() dom c
                    then where_v
                    else denote_true_t
      val max_rc = fold (op lub)(map upb_row_class
                                   (lookup_local_table_info()))
      val rcs = map denote_name(lookup_local_row_classes())
      val rvis = binop_t(dom_op,(cc,fold(curry binop_t lub_op)rcs))
      val w = if client_clearance() dom max_rc
              then wchk
              else binop_t(and_op,(wchk,rvis))
      val gbs = sterling_columns gb_sterling
      val gbd = dinary_columns gb_dinary
      val gbc = remove_constants(map class_column gb_class)

```

SML

```

val groupby_test = fold(op @)(map column_data_test
                             (gb_sterling @ gb_dinary))
val having_test = case value_class having of
                  constantec c =>
                    if client_clearance() dom c
                    then []
                    else [denote_falset]
                  | variable(exp,c) =>
                    if client_clearance() dom c
                    then []
                    else [binopt(dom_op,(cc,exp))]
val tests = groupby_test @ having_test
val checks =
  if length tests = 0
  then []
  else
    let val t = monopt(not_op,
                      (fold(curry binopt and_op)tests))
        val bad = if client_clearance() dom max_rc
                  then [where_v,t]
                  else [rvis,where_v,t]
        val junk = anonymous_valuet(denote_truet)
        val junk_sl = select_valuest[junk]
    in [selectt(all_tuplest(junk_sl,fs,
                          fold(curry binopt and_op)bad,[],denote_truet))]
    end
val sel_q = distinct_tuplest
              (sl,fs,w,gbs @ gbd @ gbc,value_data having)
in (sel_q,checks)
end)

```

SML

```

tuple_list_make_outer (evaluate_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having)
                      ,scw,to_rc,to_tcs)
= in_new_scope(fn () =>
let   val fs = map from_spec_enter fr
      val sel_vals = select_list_make(sel,to_rc,to_tcs)
      val where_v = value_data where
      val where_c = value_class where
      val where_class = case where_c of
                          variable(exp,c) => exp
                          |   constant_ec c => denote_class_t c
      val sel_list = case scw of
                      true => [anonymous_value_t where_class]@ sel_vals
                      | false => sel_vals
      val sl = select_values_t sel_list
      val cc = denote_class_t(client_clearance())
      val wchk = case where_c of
                  variable(exp,c) =>
                    if client_clearance() dom c
                    then where_v
                    else let val ok = binop_t(dom_op,(cc,exp))
                        in binop_t(or_op,(where_v,
                                      monop_t(not_op,ok)))
                        end
                  |   constant_ec c =>
                    if client_clearance() dom c
                    then where_v
                    else denote_true_t
      val max_rc = fold (op lub)(map upb_row_class
                                   (lookup_local_table_info()))
      val rcs = map denote_name(lookup_local_row_classes())
      val rvis = binop_t(dom_op,(cc,fold(curry binop_t lub_op)rcs))
      val w = if client_clearance() dom max_rc
              then wchk
              else binop_t(and_op,(wchk,rvis))
      val gbs = sterling_columns gb_sterling
      val gbd = dinary_columns gb_dinary
      val gbc = remove_constants(map class_column gb_class)

```

SML

```

val groupby_test = fold(op @)(map column_data_test
                             (gb_sterling @ gb_dinary))
val having_test = case value_class having of
                  constantec c =>
                    if client_clearance() dom c
                    then []
                    else [denote_falset]
                  | variable(exp,c) =>
                    if client_clearance() dom c
                    then []
                    else [binopt(dom_op,(cc,exp))]
val tests = groupby_test @ having_test
val checks =
  if length tests = 0
  then []
  else
    let val t = monopt(not_op,
                      (fold(curry binopt and_op)tests))
        val bad = if client_clearance() dom max_rc
                  then [where_v,t]
                  else [rvis,where_v,t]
        val junk = anonymous_valuet(denote_truet)
        val junk_sl = select_valuest[junk]
    in [selectt(all_tuplest(junk_sl,fs,
                          fold(curry binopt and_op)bad,[],denote_truet))]
    end
val sel_q = evaluatet(sl,fs,w,gbs @ gbd @ gbc,value_data having)
in (sel_q,checks)
end)

```

SML

```

| | tuple_list_make_outer (tuple_s vals,scw,to_rc,to_tcs) =
| |   let
| |     val q = denote_class_t (query_constants_class())
| |     val tvals = (case scw of
| |                 true => [q]
| |                 |   false => [])
| |                 @ (case to_rc of
| |                   constant_tc c => []
| |                   |   other => [q])
| |                 @ (fold(op @)(at2(map make_col)(vals,to_tcs)))
| |
| |     in (tuple_t tvals,[])
| |     end

```

SML

```

| | tuple_list_make_outer (union_s tls,scw,to_rc,to_tcs) =
| |   if length tls = 1
| |   then tuple_list_make_outer(hd tls,scw,to_rc,to_tcs)
| |   else let val (ts,qs) = split(at4(map tuple_list_make_outer)
| |                               (tls,
| |                                seq(length tls,scw),
| |                                seq(length tls,to_rc),
| |                                seq(length tls,to_tcs)))
| |
| |     in (union_t ts, fold (op @) qs)
| |     end

```

SML

```

| | tuple_list_make_outer (name_columns_s(names,tul),scw,to_rc,to_tcs)
| |   = tuple_list_make_outer(tul,scw,to_rc,to_tcs);

```

SML

```

| fun (tuple_list_outer_info : Tuple_list_ssql -> bool)
|   (table_contents_s t) = false

```

SML

```

| | tuple_list_outer_info(all_tuples_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having))
| |   = in_new_scope(fn () =>
| |     let
| |       val fs = map from_spec_enter fr
| |       val where_c = value_class where
| |     in
| |       case where_c of
| |         variable(exp,c) => not(client_clearance() dom c)
| |         |   constant_ec c => not(client_clearance() dom c)
| |     end)

```

SML

```

| | tuple_list_outer_info(distinct_tuples_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having))
| |   = in_new_scope(fn () =>
| |     let   val fs = map from_spec_enter fr
| |           val where_c = value_class where
| |     in   case where_c of
| |           variable(exp,c) => not(client_clearance() dom c)
| |         |   constant_ec c => not(client_clearance() dom c)
| |     end)

```

SML

```

| | tuple_list_outer_info(evaluate_s(sel,fr,where,gb_sterling,gb_dinary,gb_class,having))
| |   = in_new_scope(fn () =>
| |     let   val fs = map from_spec_enter fr
| |           val where_c = value_class where
| |     in   case where_c of
| |           variable(exp,c) => not(client_clearance() dom c)
| |         |   constant_ec c => not(client_clearance() dom c)
| |     end)

```

SML

```

| | tuple_list_outer_info(tuple_s vals) = false

```

SML

```

| | tuple_list_outer_info(union_s tls) = fold or(map tuple_list_outer_info tls)

```

SML

```

| | tuple_list_outer_info(name_columns_s(names,tul)) = tuple_list_outer_info tul;

```

SML

```

| fun (transform_select_query : Query_ssql -> SsqlCol list * Query_tsql * bool * TsqlClassName
|     * TsqlCol list * Query_tsql list)
|     (select_s vals) =
|         let   val scw = tuple_list_outer_info vals
|               val (tn,ti,scs,rc,tcs) = tuple_list_info vals
|               val (tul,chks) = tuple_list_make_outer(vals,scw,rc,tcs)
|         in   (scs,select_t tul,scw,rc,tcs,chks)
|         end
| transform_select_query q = raise internalError;

```

SML

```
| fun (query_select_query : Queryssql -> Querytsql)  
|   (selects vals) = let val (tn,ti,scs,rc,tcs) = tuple_list_info vals  
|                       in selectt(tuple_list_make(vals,rc,tcs))  
|                       end  
|  
|   query_select_query q = raise internalError;
```

### 3 INDEX

<i>all_data_columns<sub>local</sub></i> .....	20	<i>lookup<sub>local_col_spec_classes</sub></i> .....	10
<i>binop_type</i> .....	20	<i>lookup<sub>local_col_spec_sterlings</sub></i> .....	11
<i>check_boolean</i> .....	29	<i>lookup<sub>local_row_classes</sub></i> .....	11
<i>check_enum</i> .....	4	<i>lookup<sub>local_table_implementation</sub></i> .....	19
<i>check_fixed</i> .....	4	<i>lookup<sub>local_table_info</sub></i> .....	19
<i>check_floating</i> .....	4	<i>lookup<sub>param_data</sub></i> .....	19
<i>check_interval</i> .....	4	<i>lookup<sub>table_detail</sub></i> .....	14
<i>check_time</i> .....	4	<i>lookup<sub>table_row_class</sub></i> .....	13
<i>check_type_conversion<sub>domain</sub></i> .....	30	<i>lub<sub>bound_info</sub></i> .....	34
<i>check_type_conversion</i> .....	29	<i>lub<sub>col_type</sub></i> .....	35
<i>class_column</i> .....	30	<i>lub<sub>exp</sub></i> .....	35
<i>client_clearance</i> .....	4	<i>lub<sub>exp_class</sub></i> .....	34
<i>column_data_test</i> .....	31	<i>lub<sub>ssql_col</sub></i> .....	36
<i>col_exp</i> .....	31	<i>lub<sub>ssql_name</sub></i> .....	35
<i>col_target</i> .....	31	<i>lub<sub>table_info</sub></i> .....	36
<i>constant_value<sub>data</sub></i> .....	39	<i>lub<sub>tsql_class_name</sub></i> .....	36
<i>constant_value<sub>type</sub></i> .....	32	<i>lub<sub>tsql_col</sub></i> .....	36
<i>contextual_data</i> .....	4	<i>lub<sub>tsql_name</sub></i> .....	36
<i>convert<sub>col_spec</sub></i> .....	30	<i>lub<sub>type</sub></i> .....	35
<i>convert<sub>ssql_type</sub></i> .....	33	<i>lub<sub>worth</sub></i> .....	35
<i>convert<sub>sword_type</sub></i> .....	34	<i>make_col</i> .....	71
<i>convert<sub>tableSpecification</sub></i> .....	33	<i>make<sub>dinary</sub></i> .....	71
<i>convert<sub>table_spec</sub></i> .....	31	<i>make<sub>sterling</sub></i> .....	71
<i>convert<sub>type</sub></i> .....	34	<i>make<sub>sv</sub></i> .....	37
<i>default_directory</i> .....	4	<i>maxBound</i> .....	6
<i>denote_name</i> .....	30	<i>monop_type</i> .....	26
<i>denote<sub>class_exp</sub></i> .....	34	<i>query_class</i> .....	4
<i>dinary_columns</i> .....	39	<i>query_constants_class</i> .....	4
<i>enter_scope</i> .....	18	<i>query<sub>select_query</sub></i> .....	81
<i>enter<sub>corr_table</sub></i> .....	17	<i>remove_constants</i> .....	37
<i>enter<sub>identifier</sub></i> .....	15	<i>remove_nulls</i> .....	37
<i>enter<sub>identifier_constant_class</sub></i> .....	16	<i>repr_col</i> .....	19
<i>enter<sub>parameter</sub></i> .....	17	<i>select_list<sub>class</sub></i> .....	69
<i>enter<sub>table</sub></i> .....	18	<i>select_list<sub>data</sub></i> .....	71
<i>extract<sub>parameter</sub></i> .....	16	<i>select_list<sub>info</sub></i> .....	57
<i>find<sub>column</sub></i> .....	5	<i>select_list<sub>make</sub></i> .....	69
<i>find<sub>ident</sub></i> .....	6	<i>select_list<sub>type</sub></i> .....	52
<i>from_spec<sub>enter</sub></i> .....	51	<i>select_value<sub>class</sub></i> .....	69
<i>from_spec<sub>info</sub></i> .....	57	<i>select_value<sub>data</sub></i> .....	72
<i>get<sub>table_info</sub></i> .....	18	<i>select_value<sub>info</sub></i> .....	57
<i>innermost</i> .....	6	<i>select_value<sub>make</sub></i> .....	70
<i>internal_value<sub>class</sub></i> .....	58	<i>select_value<sub>type</sub></i> .....	52
<i>in_new_scope</i> .....	19	<i>set_func_type</i> .....	28
<i>leave_scope</i> .....	18	<i>simplify<sub>ands</sub></i> .....	38
<i>lookup<sub>column_info</sub></i> .....	6	<i>simplify<sub>ors</sub></i> .....	38
<i>lookup<sub>column_row_class</sub></i> .....	12	<i>sterling_columns</i> .....	39
<i>lookup<sub>col_spec_class</sub></i> .....	7	<i>table_name</i> .....	33
<i>lookup<sub>col_spec_dinary</sub></i> .....	8	<i>timeFormatToInterval</i> .....	4
<i>lookup<sub>col_spec_sterling</sub></i> .....	9	<i>transform<sub>select_query</sub></i> .....	80
<i>lookup<sub>local_col_implementation</sub></i> .....	9	<i>trip_type</i> .....	27
<i>lookup<sub>local_col_info</sub></i> .....	10	<i>tuple_list<sub>class</sub></i> .....	67

<i>tuple_list_data</i> .....	48
<i>tuple_list_info</i> .....	52
<i>tuple_list_make</i> .....	55
<i>tuple_list_make_outer</i> .....	72
<i>tuple_list_max_row_class</i> .....	39
<i>tuple_list_outer_info</i> .....	79
<i>tuple_list_type</i> .....	50
<i>unique_name</i> .....	4
<i>upb_row_class</i> .....	40
<i>upper</i> .....	37
<i>value_class</i> .....	48
<i>value_data</i> .....	40
<i>value_info</i> .....	58
<i>value_type</i> .....	43