PostgreSQL 解析資料

～ 意味解析器（アナライザ） ～

(株) NTT データ
基盤システム事業本部 オープンソース開発センタ
井久保 寛明

1. はじめに

本ドキュメントでは、意味解析器（アナライザ）による意味解析処理、および、クエリ木（Query Tree）の生成処理について解説する。意味解析器とは、構文解析器（パーザ）で作成されたパーズ木（Raw Parse Tree）に対して、意味解析を行うものである。PostgreSQLの意味解析器では、意味解析を行うと同時に、パーズ木からクエリ木を生成する。ここで生成されたクエリ木は、意味解析器以降のリライタ、プランナ、エグゼキュータの処理[1]で使用されることになる。
PostgreSQL の全てのステートメントと式は非常に膨大であるため、本ドキュメントでは、DDL 系コマンドからは CREATE TABLE 文を、DML 系コマンドからは SELECT 文を取り上げて説明することとする。また、構文解析器、リライタ、プランナ、エグゼキュータについては別のドキュメントでまとめるものとする。

1.1. 対象バージョン

本ドキュメントは、PostgreSQL7.4.3 を対象にソースコードの調査を行ったものである。従って、他のバージョンでは内容が異なる場合があるので注意して頂きたい。

1.2. ソースファイル

本ドキュメントで対象としているのは、src/backend/parser ディレクトリにあるファイルである。ここには、字句解析と構文解析に使用するファイルと意味解析に使用するファイルがある。

以下のファイルは、字句解析および構文解析処理（パーズ木生成）のソースファイルのため、本ドキュメントでは詳細な説明は行わない。

parser.c	構文解析のためのエントリポイントおよびフィルタ関数。
scan.l	字句解析器。SQL 文字列を SQL キーワードや識別子に対応するトークンの作成を行う。
scansup.c	文字列値の取り出し時の補助関数。
keywords.c	予約語を取り出す関数。

[1] エグゼキュータが使用するのは、プランナが生成したクエリプランである。しかし、クエリプランにはクエリ木へのリファレンスが張ってある部分がある。

gram.y	構文のルール（文法）を定義したファイル。bison によって生成される gram.c では、定義された文法に従って構文解析が行われ、パーズ木を生成する。

以降が、意味解析器（クエリ木生成）に関連したソースである。

analyze.c	クエリ木の生成を行う意味解析器のエントリポイントとなる関数と、各ステートメントをパーズ木からクエリ木に変換する関数。
parse_clause.c	FROM 句および､WHERE, ORDER BY, ...句等の変換に関連した関数群。
parse_coerce.c	データ型の強制(キャスト)に関連した関数群。
parse_expr.c	式(expression)ノードの変換に関連した関数群。
parse_oper.c	expression ノード中の演算子(operator)の変換に関連した関数群。
parse_agg.c	SUM(), AVG() …等、集約関数の変換に関連した関数群。
parse_func.c	関数コールの変換に関連した関数群。
parse_node.c	ParseState (Parse 情報保持領域) の確保関数と、クエリ木内で使用される幾つかのノード作成関数群。
parse_target.c	ターゲットリストの作成、操作に関連した関数群。
parse_relation.c	レンジテーブルの作成、操作に関連した関数群。
parse_type.c	データ型の操作、解析の関連した関数群。

また src/backend/nodes 以下のソースファイルについても、ノードやリストを操作するユーティリティ関数として頻繁に使用されているので参考にしてほしい。

2. クエリ処理の概要と意味解析器の位置づけ

ここでは、PostgreSQLのクエリ処理の流れと、その中における意味解析器の位置づけについて説明する。図 2-1がクエリ処理の大きな流れである。

まず、文字列のSQL文に対して字句解析と構文解析を行って、パーズ木を生成する。作成されたパーズ木に対して意味解析を行う。意味解析では、指定されたテーブルやカラムが実際にあるか確認したり、1つのSQL文を複数のSQL文に分割して実行する構文の場合のSQL文の変換を行ったりする[2]。そして、意味解析の結果としてクエリ木が生成される。次にリライト処理として、ユーザ定義のルールが定義してある場合にはクエリの変換処理を行う。リライトの結果もクエリ木である。クエリ木は、プランナ（オプティマイザ）に渡されると、論理クエリプランを生成して、最終的には1つの物理実行プランが書かれたプラン木を生成する。このときにパスツリーと呼ばれる、同じ実行結果になる複数の異なる実行プランが生成され、その中で最適なものをコストベースで選択する。最終的に選択されたパスツリーをプラン木に変換し、そのプラン木をクエリエグゼキュータに渡してSQLの処理を実行する。

[2] 厳密には、SQL文を分割するのではなく、パーズ木を分割する。

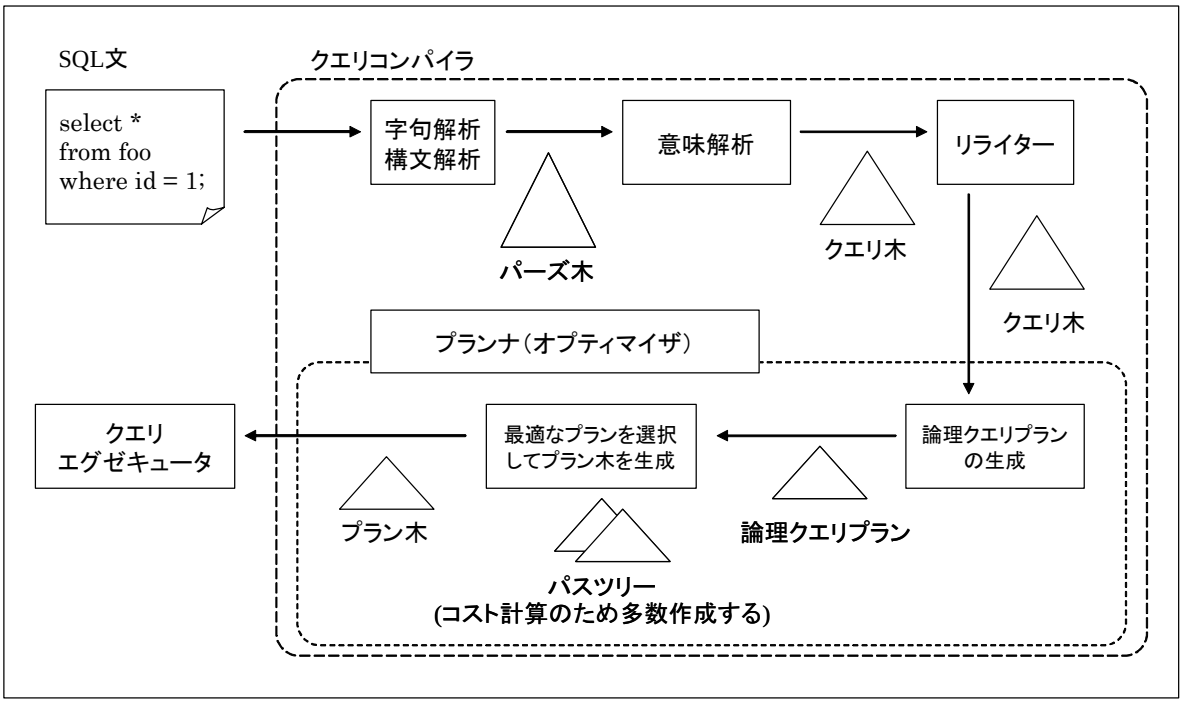

図 2-1 クエリ処理の流れ

パーズ木とクエリ木は、非常によく似た構造をしている。クエリ木は、パーズ木の不足を補って、後の処理に使いやすい形式に変換したものだと考えてよい。DBMS によっては、これらを区別しないものもある。

構文解析と意味解析を分ける理由は、DB へのアクセスを減らすためである。構文解析では、構文規則に従ったエラーのチェックのみを行う。この段階でエラーと判定できるものに関しては、DB へのアクセスを必要としない。意味解析でテーブルやカラムなどの名称が実際に DB に登録されているかどうかを調べる。つまり、この段階ではじめて DB へのアクセスを行う。このように、構文解析と意味解析を分けることで、エラーになる処理で DB へアクセスする機会を減らしている。

3. パーズ木の構造

意味解析器の処理内容およびクエリ木の構造を理解するには、変換元となるパーズ木の構造についてある程度理解しておく必要があるため、ここで簡単に説明する。字句解析器と構文解析器は、文字列の SQL 文から、パーズ木を生成する。パーズ木は、SQL 構文の構造をそのまま木構造で表現したような形式になっている。木構造のトップは、各 SQL ステートメントの構造体であり、これらは src/include/nodes/parsenodes.h で定義されている。ステートメント構造体の種類は多岐に渡るため、ここでは、CREATE TABLE ステートメントの構造体 CreateStmt と、SELECT ステートメントの構造体 SelectStmt について説明する。

3.1. CREATE TABLE ステートメント構造体 CreateStmt

CREATE TABLE ステートメント構造体 CreateStmt は次のように定義されている。

```
typedef struct CreateStmt
{
  NodeTag           type;           /* Node の Tag */
  RangeVar          *relation;      /* リレーション情報 */
  List              *tableElts;     /* カラム定義 (ColumnDef の List) */
  List              *inhRelations;  /* 継承するリレーションの List 継承関係 (INHERITS 句) */
  List              *constraints;   /* CHECK 制約のリスト (Constraint の List) */
  bool              hasoids; /* OID を持っているか (WITH OIDS / WITHOUT OIDS) */
  OnCommitAction    oncommit;/* COMMIT のアクション (ON COMMIT) */
} CreateStmt;
```

type フィールドには、CREATE TABLE ステートメントのノードを表すタグ T_CreateStmt がセットされる。タグの種類は include/nodes/nodes.h で定義されている。

relation フィールドには "CREATE TABLE *table_name* ..." の *table_name* に相当する作成テーブルのリレーション情報 RangeVar ノードが格納される。

tableElts フィールドには、"CREATE TABLE *table_name* (... , ...)" の "(... , ...)" に相当するカラム定義やテーブル制約等の情報がリストで格納される。カラム定義の場合は ColumnDef ノード、テーブル制約が定義されている場合には制約情報 Constraint ノードでリストが構成される。なお、カラム定義情報 ColumnDef ノードは、データ型情報 TypeName ノードや制約情報 Constraint ノードのリストとして構成される。

INHERITS 句が定義されている場合は、inhRelations フィールドに継承するリレーション情報 RangeVar ノードのリストが格納される。

constraints フィールドはパーズ木の生成段階ではまだ使用されない。

実際のCREATE TABLE文を例にしたパーズ木の構造を、図 3-1に示す。

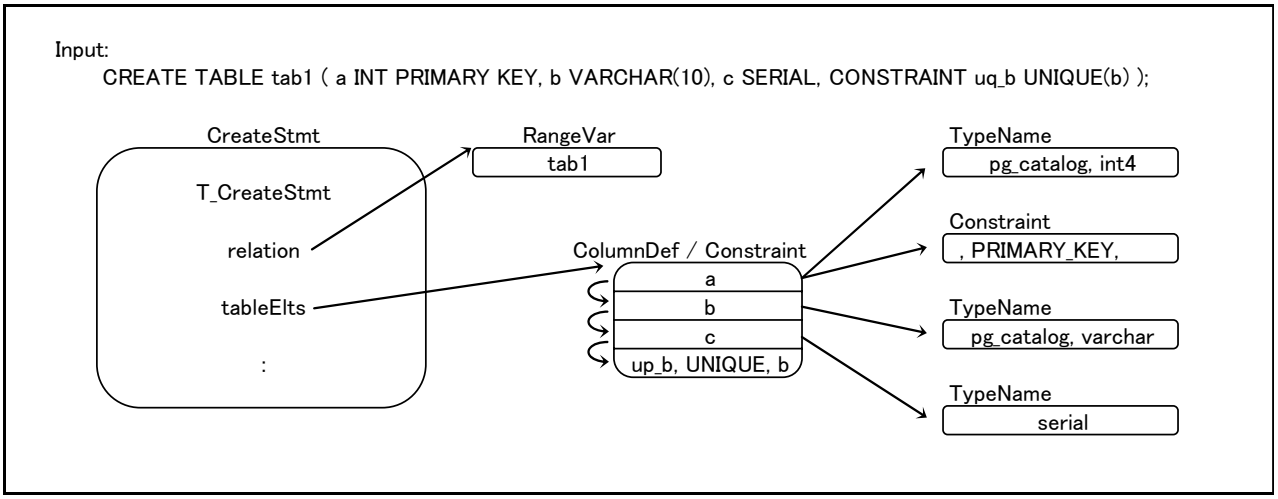

図 3-1 CreateStmt のパーズ木

3.2. SELECT ステートメント構造体 SelectStmt

SELECT ステートメント構造体 SelectStmt は次のように定義されている。

```
typedef struct SelectStmt
{
  NodeTag           type;             /* Node の Tag */

  /*
   * These fields are used only in "leaf" SelectStmts.
   * into and intoColNames are a kluge; they belong somewhere else...
   */
  List     *distinctClause;   /* DISTINCT ON や SELECT DISTINCT のためのリスト */
  RangeVar *into;             /* SELECT INTO / CREATE TABLE AS の(格納先の)リレーション情報 */
  List     *intoColNames;     /* 上記のカラム名リスト (ColumnDef のリスト) */
  List     *targetList;       /* ターゲットリスト (ResTarget のリスト) */
  List     *fromClause;       /* FROM 句のリスト */
  Node     *whereClause;      /* WHERE 句の条件ノード */
  List     *groupClause;      /* GROUP BY 句のリスト */
  Node     *havingClause;     /* HAVING 句の条件ノード */

  /*
   * These fields are used in both "leaf" SelectStmts and upper-level SelectStmts.
   */
  List     *sortClause;       /* SORT (ORDER BY 句)のリスト (SortBy のリスト) */
  Node     *limitOffset;      /* LIMIT OFFSET のノード */
  Node     *limitCount;       /* LIMIT COUNT のノード */
  List     *forUpdate;        /* FOR UPDATE 句のリスト */

  /*
   * These fields are used only in upper-level SelectStmts.
   */
  SetOperation      op;       /* 集合演算子のタイプ (NONE/UNION/INTERSECT/EXCEPT) */
  bool              all;      /* ALL オプションが指定されているか */
  struct SelectStmt *larg;    /* 左の SelectStmt ステートメント */
  struct SelectStmt *rarg;    /* 右の SelectStmt ステートメント */
} SelectStmt;
```

type フィールドには、SELECT ステートメントのノードを表すタグ T_SelectStmt がセットされる。
SELECT句で定義される、問い合わせの結果カラム名[3]のリストは「ターゲットリスト」と呼ばれる。
パーズ木においては、ターゲットリストはResTargetノードのリストで構成され、targetListフィールドに格納される。ResTargetノード内は、カラム参照情報(ColumnRef) や関数コール(FuncCall)、演算(A_Expr)、サブクエリ(SubLink)等のexpressionノードで構成される。なおパーズ木の段階では、"SELECT * FROM boo" のような、"*" のカラム情報の展開処理 ("SELECT foo, woo FROM boo" に展開) は行わない。
FROM 句に定義されるリストは、「レンジテーブル」と呼ばれ、fromClause フィールドに格納される。レンジテーブルには、それぞれのテーブルに対応したエイリアス名、リレーション名を持つリレーション情報 RangeVar ノードや、JOIN(JoinExpr)や FROM 句中のサブクエリ(RangeSubselect)等の expression ノードのリストで構成される。
WHERE 句を表現するツリーは、関数コール(FuncCall)や演算(A_Expr)、サブクエリ(SubLink)等の expression ノードで構成され、whereClause フィールドに格納される。

[3] 演算なども書くことができる。

into および intoColNames フィールドは、SELECT INTO 文や CREATE TABLE AS 文で使用される。これらは専用のステートメント構造体が存在せず、SelectStmt 構造体でパーズ木が生成されるので注意してほしい。
op フィールドから rarg フィールドまでは、集合演算(問い合わせの結合)の際に使用される。op フィールドには結合演算子のタイプが格納され、その種類には NONE / UNION / INTERSECT / EXCEPT があり、集合演算無しの場合は NONE がセットされる。集合演算ありの場合は、左辺と右辺の SELECT ステートメントノードが、larg と rarg フィールドに格納される。

実際のSELECT文を例にしたパーズ木の構造を、図 3-2に示す。

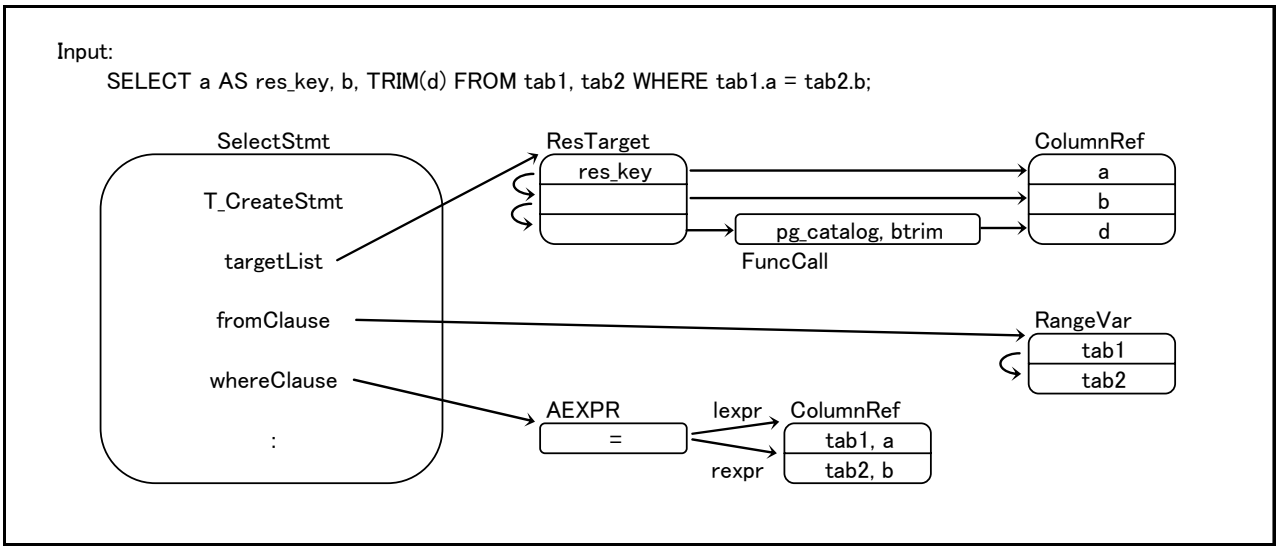

図 3-2 SelectStmt のパーズ木

ここまでの説明では、式を表現するノードを expression ノードとして、ある程度一まとめに説明しているが、これらは src/include/nodes/parsenodes.h と src/include/nodes/primnode.h に定義されている。expression ノード内には、さらに幾つかの expression ノードで（場合によって再帰的に）構成され、木構造となる。各 expression ノード構造体の種類は多岐に渡るため、本ドキュメントでは全てを取り上げず、必要に応じて説明することとする。

簡単なexpressionノードの構成を、サブクエリとJOINを例にして、図 3-3に示す。

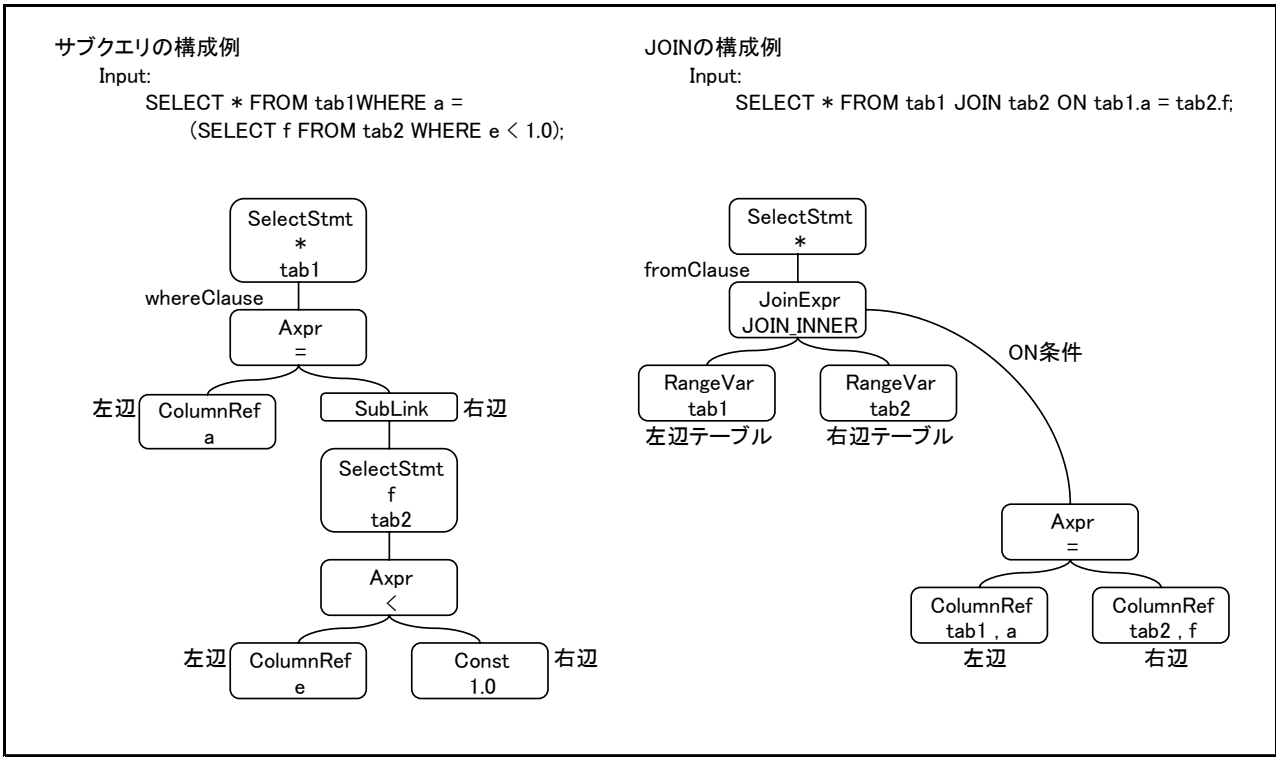

図 3-3 expression ノードの例

図中のサブクエリの構成例では、WHERE 句にサブクエリが書かれているケースで、whereClause フィールド内にサブクエリを表現したサブツリーが作成されている。FROM 句にサブクエリがある場合は fromClause に、SELECT 句にサブクエリがある場合は tagetList フィールドに、サブクエリを表現したサブツリーが作成される。

3.3. その他のステートメント構造体

PostgreSQLには、CreateStmtとSelectStmt以外のステートメント構造体が多数存在するが、基本的にリレーションはRangeVarで[4]、カラム定義[5]はColumnDef、カラム参照[6]はColumnRefで、その他expressionノードで構成される点については同様である。

4. クエリ木の構造

パーザによって生成されたパーズ木は、意味解析器によってクエリ木に変換される。この章では、クエリ木の構造について説明する。

クエリ木の各要素は、パーズ木と構造的には似ているが、実際は多くの相違点がある。ステートメントの種類にもよるが、パーズ木で文字列表現されているリレーションやデータ型、関数、演算子などをシステムカタログにアクセスして実際の OID に変換したり、演算式におけるデータ型合わせを行ったり、文字列表現されている定数値の実データ型への値変換をする。このように、多くのノードやノードを構成する各要素が、クエリ木用に書き換えられる。

[4] クエリ木に変換された後は、リレーション定義とリレーション参照とでは異なる構造体で表現される。
[5] CREATE文などで、テーブルを作成するときのカラムの制約情報の保持などに使用される。
[6] すでに定義されているカラム情報への参照。

ちなみに、意味解析直後のクエリ木は、debug_print_parse 設定パラメータを設定することによって、サーバログに出力することができる。同様にリライト後のクエリ木は debug_print_rewritten で出力することができる。

4.1. Query 構造体

クエリ木のルートは、Query ノードとなる。Query ノードを構成する Query 構造体は src/include/nodes/parsenodes.h において、次のように定義されている。

```
typedef struct Query
{
  NodeTag           type;           /* Node の Tag */
  CmdType           commandType;    /* コマンドタイプ (unknown | select | insert | update |
                                       delete | utility | nothing) */
  QuerySource       querySource;    /* どのように生成されたか？ */
  bool              canSetTag;      /* リザルトタグをセットできるかどうかのフラグ */
  Node              *utilityStmt;   /* DDL やユーティリティ系(DML 以外)のステートメント */
  int               resultRelation; /* 関係する rtable へのインデックス */
  RangeVar          *into;          /* SELECT INTO のためのリレーション情報 */
  bool              hasAggs;        /* targetList、あるいは havingQual で集約を持って
                                       いるかのフラグ */
  bool              hasSubLinks;    /* サブクエリへのリンクを持っているかのフラグ */
  List              *rtable;        /* レンジテーブル。RangeTblEntry のリスト */
  FromExpr          *jointree;      /* FROM 句 や WHERE 句のリレーション結合情報と問い合わせ
                                       条件ノード */
  List              *rowMarks;      /* FOR UPDATE リレーションのインデックスリスト */
  List              *targetList;    /* ターゲットリスト。TargetEntry のリスト */
  List              *groupClause;   /* GROUP 句のリスト */
  Node              *havingQual;    /* グループに適用される条件(制限) */
  List              *distinctClause; /* DISTINCT 句のリスト */
  List              *sortClause;    /* SORT(ORDER BY 句)のリスト */
  Node              *limitOffset;   /* LIMIT OFFSET のノード */
  Node              *limitCount;    /* LIMIT COUNT のノード */
  Node              *setOperations; /* 集合(UNION/INTERSECT/EXCEPT)を表現するノード */
  List              *resultRelations; /* リザルト(結果)リレーション (プランナでセット) */
  /* internal to planner */
  List              *base_rel_list; /* list of base-relation RelOptInfos */
  List              *other_rel_list; /* list of other 1-relation RelOptInfos */
  List              *join_rel_list; /* list of join-relation RelOptInfos */
  List              *equi_key_list; /* list of lists of equijoined PathKeyItems */
  List              *in_info_list;  /* list of InClauseInfos */
  List              *query_pathkeys; /* desired pathkeys for query_planner() */
  bool              hasJoinRTEs;    /* true if any RTEs are RTE_JOIN kind */
} Query;
```

type フィールドにはノードのタグがセットされる。意味解析器がセットするタグは T_Query のみである。

commandType フィールドには、その Query ノードが表現しているステートメントのコマンドタイプがセットされる。コマンドタイプには以下の種類がある。

CMD_UNKNOWN	未知のコマンドタイプ。実装上ありえない。
CMD_SELECT	SELECT ステートメント。

CMD_INSERT	INSERT ステートメント。
CMD_UPDATE	UPDATE ステートメント。
CMD_DELETE	DELETE ステートメント。
CMD_UTILITY	SELECT / INSERT / UPDATE / DELETE 以外のステートメント。 DDL 系コマンドや、VACUUM 等のユーティリティコマンドの場合。
CMD_NOTHING	CREATE RULE で action 指定がない場合のダミー用。

querySource フィールドには、その Query ノードがどのように作成されたかの種別がセットされる。以下にその種別をあげる。

QSRC_ORIGINAL	パーザから受け取ったステートメントに相当するオリジナルの Query ノード。
QSRC_PARSER	意味解析器が拡張あるいは分割した、追加 Query ノード。
QSRC_INSTEAD_RULE	ルール(条件無し INSTEAD ルール)で追加された場合。
QSRC_QUAL_INSTEAD_RULE	ルール(条件付き INSTEAD ルール)で追加された場合。
QSRC_NON_INSTEAD_RULE	ルール(INSTEAD 無しルール)で追加された場合。

canSetTag フィールドは、問い合わせ結果やコマンド実行結果をセットできるかどうかのフラグがセットされる。
utilityStmtフィールドは、コマンドタイプ（commandType）がCMD_UTILITYの場合にのみ使用され、意味解析器で変換した後[7]のステートメントノードが格納される。
resultRelation フィールドには、問い合わせ結果が格納されるリレーションを識別するための、レンジテーブルのインデックスが格納される。SELECT 文の場合は使用されない(ただし SELECT INTO 文は除く)。INSERT、UPDATE、DELETE 文の場合は、対象となるリレーションを示す。
rtableフィールドには、意味解析器で変換された後の、問い合わせで使用されるリレーション参照情報を取りまとめたリスト(レンジテーブル)が格納される。レンジテーブルについては **4.1.1レンジテーブル**で説明する。
jointreeフィールドには、リレーションの結合構造を表現するRTE[8]参照ノードのリストとWHERE句の問い合わせ条件のノードが格納される。
targetListフィールドには、問い合わせ結果を定義する情報を取りまとめたリスト(ターゲットリスト)が格納される。ターゲットリストについては**4.1.2 ターゲットリスト**で説明する。

DML ステートメントの Query ノード生成処理は、前記のレンジテーブル(rtable)、リレーション結合構造および問い合わせ条件（jointree)、ターゲットリスト（targetList）の変換処理が主体となる。
一方、DDL やユーティリティ系ステートメントのトップノードの Query ノードでは、これらはほとんど作成されない。代わりに、パーズ木で作成されたステートメントノードの変換が主体で、変換されたステートメントノードが utilityStmt フィールドに格納されることになる。

[7] ステートメントの種類によっては、パーザから受け取った状態の未変換のステートメントノードがそのまま格納される。
[8] レンジテーブルエントリのこと。

4.1.1. レンジテーブル

レンジテーブルは、問い合わせで使用されるリレーション参照情報を表現するRangeTblEntry（**RTE**[9]）ノードのリストで構成されている。意味解析器ではパーズ木のレンジテーブルを認識した場合に、システムカタログ(pg_class)に存在することを確認し、リレーション名やエイリアス名、リレーションのOIDを持つRTEノードを作成する。RTEの作成中にシステムで認識できないリレーションが見つかった場合には、エラーを返して処理をアボートする。

RangeTblEntry 構造体は src/include/nodes/parsenodes.h において次のように定義されている。

```
typedef struct RangeTblEntry
{
  NodeTag           type;
  RTEKind           rtekind;        /* RangeTblEntry の種別 */
  Oid               relid;          /* リレーションの OID */
  Query             *subquery;      /* サブクエリ */
  Node              *funcexpr;      /* ファンクションコールのためのツリー */
  List              *coldeflist;    /* ColumnDef ノードのリスト */
  JoinType          jointype;       /* JOIN のタイプ */
  List              *joinaliasvars; /* JOIN の Var リスト */
  Alias             *alias;         /* ユーザ定義されたエイリアス (AS xxx) */
  Alias             *eref;          /* 意味解析器によって生成(拡張)されたエイリアス */
  bool              inh;            /* 継承を要求しているか */
  bool              inFromCl;       /* FROM 句中か  */
  bool              checkForRead;   /* リレーションの読み込みアクセスチェックを行なうか */
  bool              checkForWrite;  /* リレーションの書き込みアクセスチェックを行なうか */
  Oid               checkAsUser;    /* アクセス権限チェックの対象ユーザ */
} RangeTblEntry;

typedef enum RTEKind
{
  RTE_RELATION,     /* 通常のリレーション参照 */
  RTE_SUBQUERY,     /* FROM 中のサブクエリ */
  RTE_JOIN,         /* JOIN */
  RTE_SPECIAL,      /* 特別のルールリレーション (NEW or OLD) */
  RTE_FUNCTION      /* FROM 中のファンクション */
} RTEKind;
```

RTEは、通常のリレーション参照や、サブクエリの結果セットのリレーション[10]、テーブル関数呼び出しのリレーション[11]、あるいはJOINの対象となるリレーション情報等を表現している。これらはrtekindフィールドにRTEKindで格納される。

rtekind が RTE_SUBQUERY の場合には、subquery フィールドにそのサブクエリの Query ノードが格納される。RTE_JOIN の場合には、jointype フィールドに JOIN の種別が、joinaliasvars フィールドには結合するカラム情報を表現した、レンジテーブル内のリレーションの位置とカラム位置を持つ Var ノードのリストが格納される。RTE_FUNCTION の場合には、funcexpr フィールドにテーブル関数呼び出しのノードが格納される。

alias フィールドには、FROM 句で *tablename* AS として作成したテーブルの別名情報が、Alias ノー

[9]以降、本ドキュメントではレンジテーブルエントリをRTEという略称で扱う。ソース中のコメント内でもRTEと略されている場合が多い。

[10] サブクエリの結果セットのリレーションとは、SELECT文のFROM句内のサブクエリや、INSERT ...SELECT ... のように、問い合わせ結果セットを一つのテーブルとして使用する場合を指す。

[11] テーブル関数呼び出しのリレーションとは、SELECT文の場合、FROM句中の関数呼び出しのことである。

ドとして格納される。
erefフィールドには、意味解析器で拡張されたリレーション名とカラム名リストの情報が、Aliasノードとして格納される。リレーション名は通常テーブル名がそのまま使用されるが、RTE_RELATION以外の場合は、ある一定の命名規則によって名前付けされる[12]。カラム名リストには対象となるリレーション中の全カラム名が格納される。
checkForRead、checkForWrite フィールドは、エグゼキュータ実行時のリレーションへの Read/Write アクセスチェックの有無フラグと思われるが、どのように使われるかは今回調査を行なっていない。
基本的に SELECT 時には checkForRead に true がセットされ、INSERT や DELETE、UPDATE 時には checkForWrite に true がセットされる。

なお、レンジテーブルはリレーションの参照を表現するものであるため、CREATE TABLE や DROP TABLE、ALTER TABLE などの DDL ステートメントでは作成されない。
レンジテーブルの作成や操作に関わるソースは、主に src/backend/parser/parse_relation.c に取りまとめられている。

4.1.2. ターゲットリスト

ターゲットリストは、問い合わせの結果の定義情報TargetEntry[13]ノードのリストで構成されている。
意味解析器では、パーズ木のターゲットリストや暗黙のターゲット[14]を認識した場合に、問い合わせ結果のカラム名や位置を表現したResdomノードと、レンジテーブル内のリレーションの位置とカラム位置を持つVarノード、もしくは関数呼び出し、定数、演算子、サブクエリ等を表現する各expressionノードで構成されたTargetEntryを作成する。
パーズ木で文字列表現されていたデータ型や関数、演算子などは、実際の OID に変換される。また文字列表現の定数値も実データ型の値に変換される。これらは TargetEntry の作成中にシステムカタログ(pg_type / pg_proc / pg_operator / pg_cast など)にアクセスして変換するので、システムカタログ中のタプルに見つからない場合はエラーを返して処理をアボートする。
同様に、レンジテーブル中に存在しないリレーション名や属性名（カラム名）が見つかった場合には、エラーを返して処理をアボートする。

SELECT 文の場合の TargetEntry は、SELECT 句に書かれている問い合わせの最終結果を構成するための要素で、結果の並び順とリレーション、カラムの特定(位置)情報、および式(関数、定数、演算子、サブクエリ等)ノードで表現される。
INSERT 文の場合の TargetEntry は VALUES()を構成するための要素で、INSERT 対象のカラムの特定(位置)情報、および式(関数、定数、演算子、サブクエリ等)ノードで表現される。ただし INSERT ... SELECT の場合の INSERT 句の TargetEntry は、SELECT 句用の Query ノードの TargetEntry をコピーしたものになる。
UPDATE 文の場合は SET col = value, …の SET 内を構成するための要素で、UPDATE 対象のカラムの特定(位置)情報、および式(関数、定数、演算子、サブクエリ等)ノードで表現されることとなり、

[12] 例えばJOINの場合は “unnamed_join” という名前が付けられる。
[13] 一部のソース中のコメント内では、TEと略されている場合がある。
[14] 明示的なターゲットとは、SELECT句に書かれたターゲットである。暗黙のターゲットリストとは、SELECT句に現れないが、ターゲットリストとして必要なものである。例えば、WHERE句やORDER BY句だけに現れるようなものである。

INSERT 文の場合と同等である。

なお、DELETE 文の場合はターゲットリストを作成しない。また、レンジテーブルと同様に、CREATE TABLE や DROP TABLE、ALTER TABLE などの DDL ステートメントでもターゲットリストを作成しない。

ターゲットリストの作成や操作に関わるソースは、主に src/backend/parser/parse_target.c にまとめられている。

TargetEntry 構造体は include/nodes/primnodes.h において次のように定義されている。

```
typedef struct TargetEntry
{
  Expr              xpr;
  Resdom            *resdom; /* descriptor for targetlist item */
  Expr              *expr;           /* expression to evaluate */
} TargetEntry;
```

resdom フィールドは、問い合わせ結果のカラム名や位置を表現した Resdom ノードが格納される。
expr フィールドには、ターゲットが属性(カラム)の場合、レンジテーブル内のリレーションの位置とカラム位置を持つ Var ノードが格納される。ターゲットが関数呼び出し、定数、演算子等の式で記述されている場合には、それらを表現する各 expression ノードが格納される。

TargetEntry を構成する Resdom ノードの構造体は次のように定義されている。

```
typedef struct Resdom
{
  NodeTag           type;
  AttrNumber        resno;          /* 結果カラムの番号(位置) */
  Oid               restype;        /* 結果カラムの型 OID */
  int32             restypmod;      /* 結果カラムの型 typmod の値 */
  char              *resname;       /* 結果カラム名 */
  Index             ressortgroupref; /* SORT/GROUP により参照されている場合の Index */
  Oid               resorigtbl;     /* リレーション OID */
  AttrNumber        resorigcol;     /* カラム番号(位置) */
  bool              resjunk;        /* true の場合、結果出力に必要のないカラム情報
                                       (例えばソートキー)*/
} Resdom;
```

resno フィールドの使われ方は、SELECT 文とそれ以外では異なる。
SELECT 文の場合、resno は問い合わせ結果の並び順を示す位置となる。(1 から数える)
INSERT あるいは UPDATE 文の場合、resno はリレーション内のカラム番号(位置)を示しており、対象となるカラムを特定している。
resorigtbl と resorigcol フィールドにはリレーション OID とリレーション内のカラム位置が格納されるが、SELECT 文の通常リレーションの表現以外では使用されない。

SELECT 文の問い合わせ結果項目が単純なカラム指定の場合、TargetEntry の expr フィールドは Var ノードが格納される。この場合の Var ノードは、問い合わせ結果項目 Resdom に紐付くレンジテーブル内のリレーション番号(位置)、カラム番号(位置)で構成されていて、問い合わせの最終結果のリレーションとカラムを特定している。

Var ノードの構造体の定義は以下のとおりである。

```
typedef struct Var
{
  Expr              xpr;
  Index             varno;          /* レンジテーブルのインデックス (1 から) */
  AttrNumber        varattno;/* 属性(カラム)番号。0 の場合は全て */
  Oid               vartype; /* 型 OID */
  int32             vartypmod;      /* 型 typmod の値 */
  Index             varlevelsup;

  /*
   * for subquery variables referencing outer relations; 0 in a normal
   * var, >0 means N levels up
   */
  Index             varnoold;/* original value of varno, for debugging */
  AttrNumber        varoattno;      /* original value of varattno */
} Var;
```

レンジテーブルとターゲットリストの関係を、以下の図で示す。
図 4-1がSELECT文を例にした構成で、図 4-2はUPDATE文およびINSERT文を例にした構成となっている。

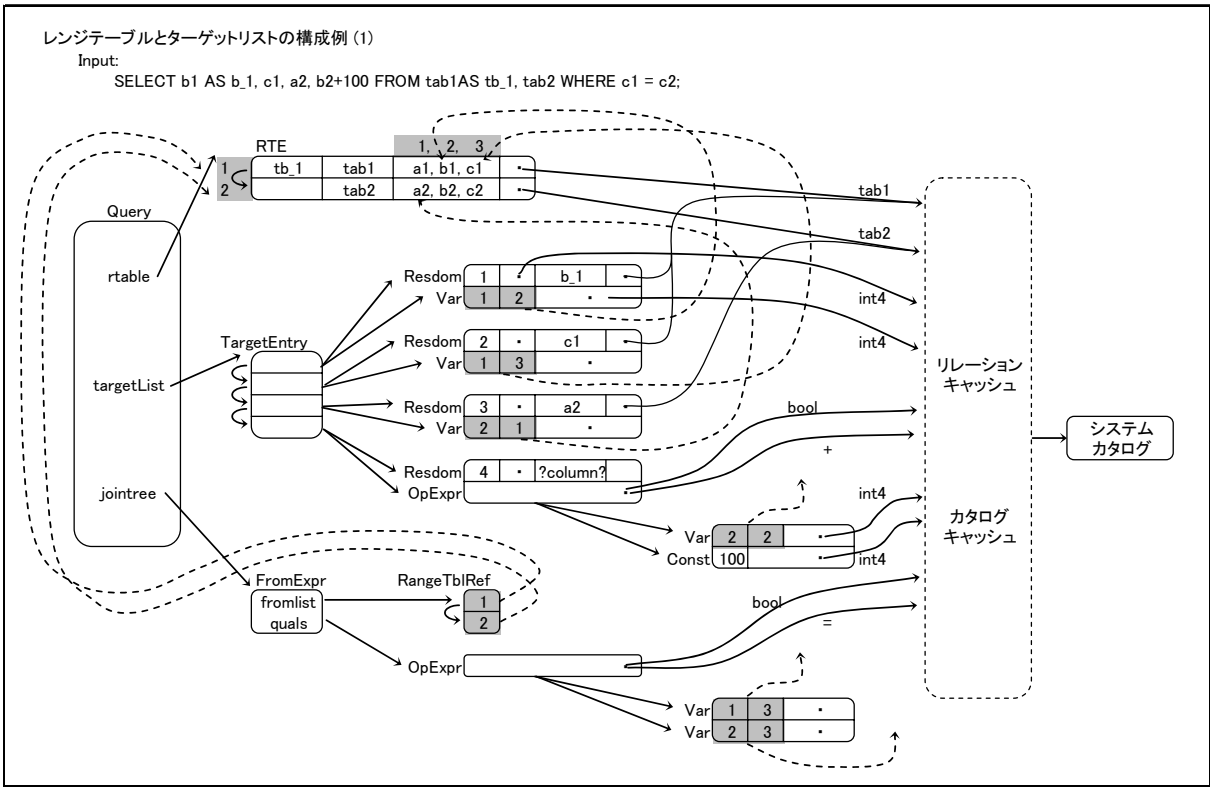

図 4-1 レンジテーブルとターゲットリスト（SELECT 文）

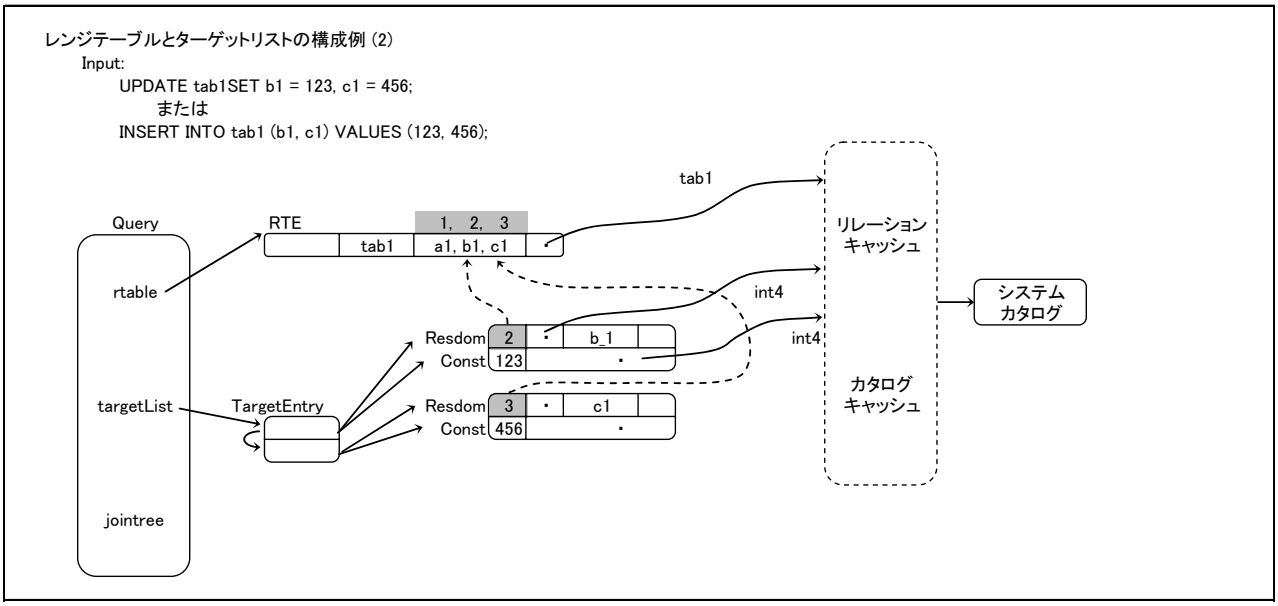

図 4-2 レンジテーブルとターゲットリスト（UPDATE 文と INSERT 文）

5. 意味解析器によるクエリ木生成処理

意味解析器は、構文解析器からパーズ木を受け取り、その中を再帰的に解析しながら Query ノードをトップツリーとするクエリ木を生成する。この章では、パーズ木からクエリ木への変換について説明する。

5.1. 意味解析器の処理概要

意味解析器のエントリポイントはparse_analyze() 関数である。parse_analyze()関数の処理シーケンスを図 5-1に示す。

parse_analyze() 関数では、後で説明するパーズ情報保持領域(ParseState)を確保する。実際にレンジテーブルやターゲットリストを作成したり、式を解析して各ノードを変換したりするのは、transformStmt() 関数に入ってからである。

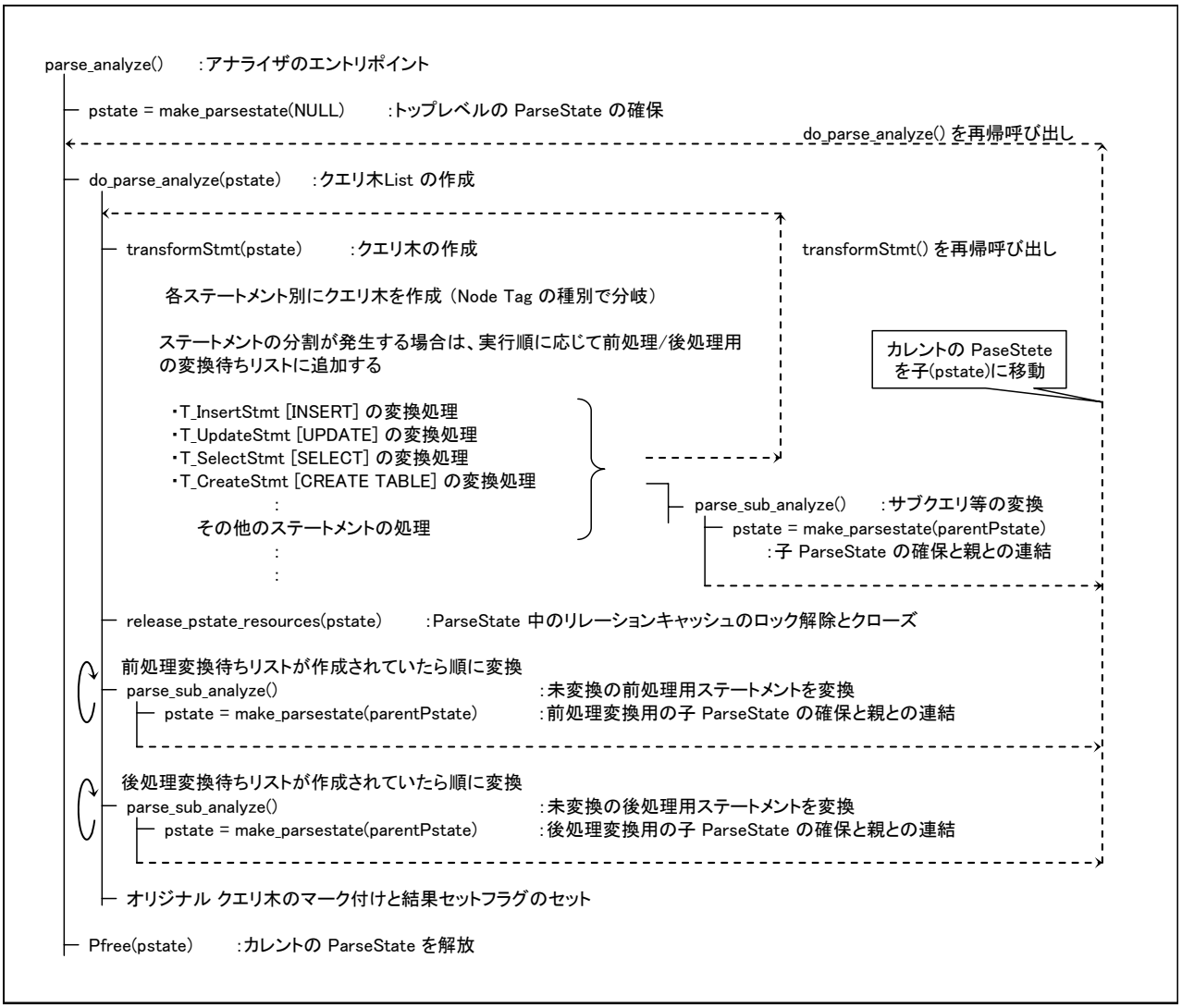

図 5-1 parse_analyze()関数の処理シーケンス

transformStmt() 関数では、各ステートメント別に分岐して変換処理を実行したり、ステートメント別のクエリ木変換関数を呼び出したりする。つまり、transformStmt() 関数は、ステートメント変換のエントリポイントとなっていて、各ステートメントの変換関数ではステートメントに応じた意味解析が行なわれる。意味解析では、レンジテーブルやターゲットリストの作成、あるいは式を解析して様々な expression ノードに変換していく。expression ノード変換のエントリポイントは、transformExpr() 関数となっており、これらの関数が再起的に呼び出されながら変換処理が進む。

なお、ステートメントの内容によっては、１つのパーズ木の意味解析を行った結果、複数のクエリ木に分割される場合がある[15]。そのためparse_analyze() 関数では最終的にクエリ木のリストを返す仕様となっている。本来のステートメントの実行を達成するために、本筋の処理を意味するオリジナルのクエリ木と、その前処理を行うクエリ木、後処理を行うクエリ木に分割され、実行順にリスト形式で配置される。
transformStmt() 関数によって分割された前処理、後処理は、まだクエリ木としての変換が終わっていない。この未変換の前処理、後処理は、do_parse_analyze()関数を再帰呼び出しすることによってク

[15] 実際に分割されるステートメントは、CREATE TABLE文とALTER TEBLE文だけのようである。

エリ木に変換される。再帰呼び出しは未変換の前処理、後処理が発生しなくなるまで続けられ、クエリ木のリストを構築していく。

前処理、後処理に分割されたステートメントの変換手順を図 5-2で、クエリ木リストの構造とCREATE文でのステートメント分割の例を図 5-3に示す。

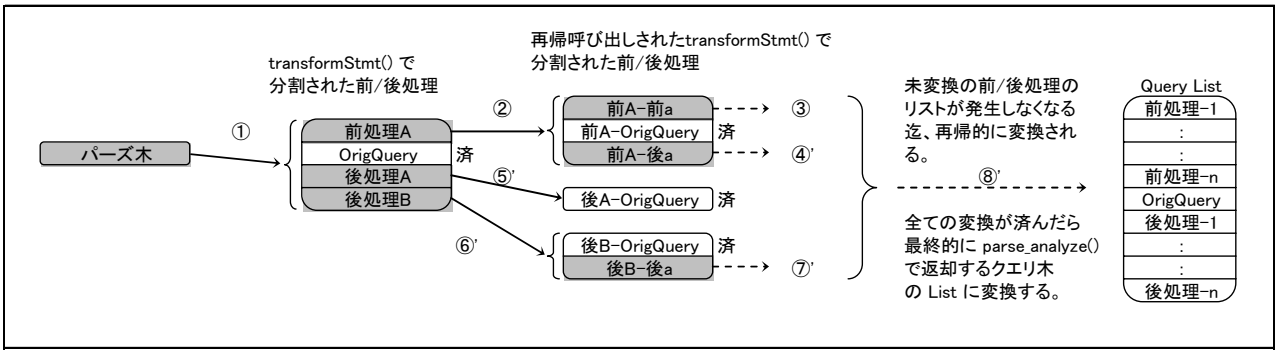

図 5-2 前処理、後処理に分割されたステートメントの変換手順

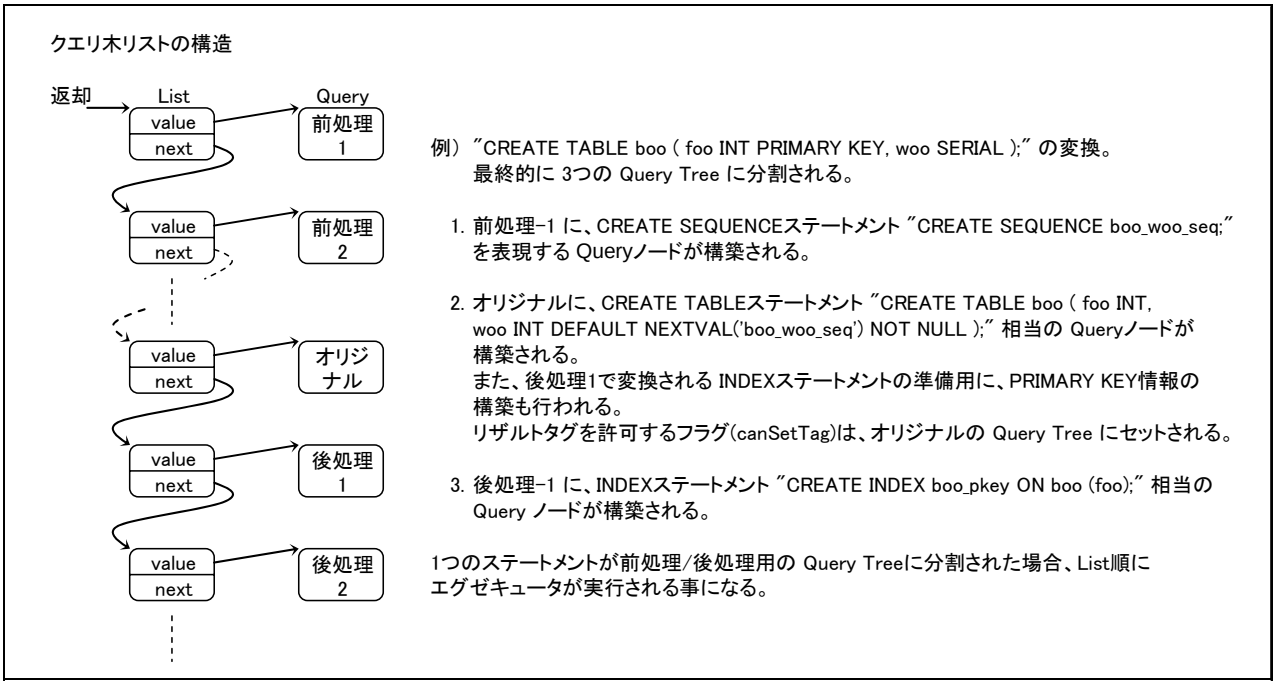

図 5-3 クエリ木リストの構造と CREATE 文でのステートメント分割の例

5.2. パーズ情報保持領域 – ParseState 構造体

意味解析中は、パーズ情報保持領域(ParseState)に、リレーション情報等の各種情報を随時保持、更新する。そして、parse_analyze()関数が終了するまで続く変換処理で活用される。

ParseState 構造体は、include/parser/parse_node.h において次のように定義されている。

```
typedef struct ParseState
{
  struct ParseState  *parentParseState; /* 親のパーズ情報保持領域へのリンク */
  List               *p_rtable;                 /* レンジテーブルとなる RTE のリスト */
  List               *p_joinlist;               /* FROM や WHERE 句のテーブルの結合構造
                                                /* (FROM 句の構造を表現するノード) */
  List               *p_namespace;              /* RTE 探索用のネームスペースのリスト */
  Oid                *p_paramtypes;             /* n パラメータシンボルのための型の OID */
                                                /* exec_parse_message()から CALL される場合の */
                                                /* 埋め込み SQL の PREPARE で使用される */
  int                p_numparams;               /* p_paramtypes のアロケートサイズ */
                                                /* exec_parse_message()から CALL される場合の */
                                                /* 埋め込み SQL の PREPARE で使用される */
  int                p_next_resno;              /* 次に割り当て予定のターゲットリストの resno */
  List               *p_forUpdate;              /* FOR UPDATE 句のリスト */
  Node               *p_value_substitute;       /* あれば replace VALUE */
  bool               p_variableparams;          /* パラメータ付きクエリ(prepared stmt)か */
  bool               p_hasAggs;                 /* 集約を持つか */
  bool               p_hasSubLinks;             /* サブクエリを持つか */
  bool               p_is_insert;               /* INSERT か */
  bool               p_is_update;               /* UPDATE か */
  Relation           p_target_relation;         /* リレーションキャッシュ情報(ターゲットリレー
                                                /* ション) */
  RangeTblEntry      *p_target_rangetblentry;   /* この ParseState で取り扱った最新の RTE */
} ParseState;
```

サブクエリが認識されて変換を行なう場合や、ステートメントが分割されて前処理や後処理を意味解析処理する場合などは、新たに子の ParseState が確保される。必要に応じて親へのリンクを辿ることにより、上位レベルの意味解析情報も共用できるような仕組みとなっている。
意味解析処理実行中のParseStateの階層構造を図 5-4で説明する。

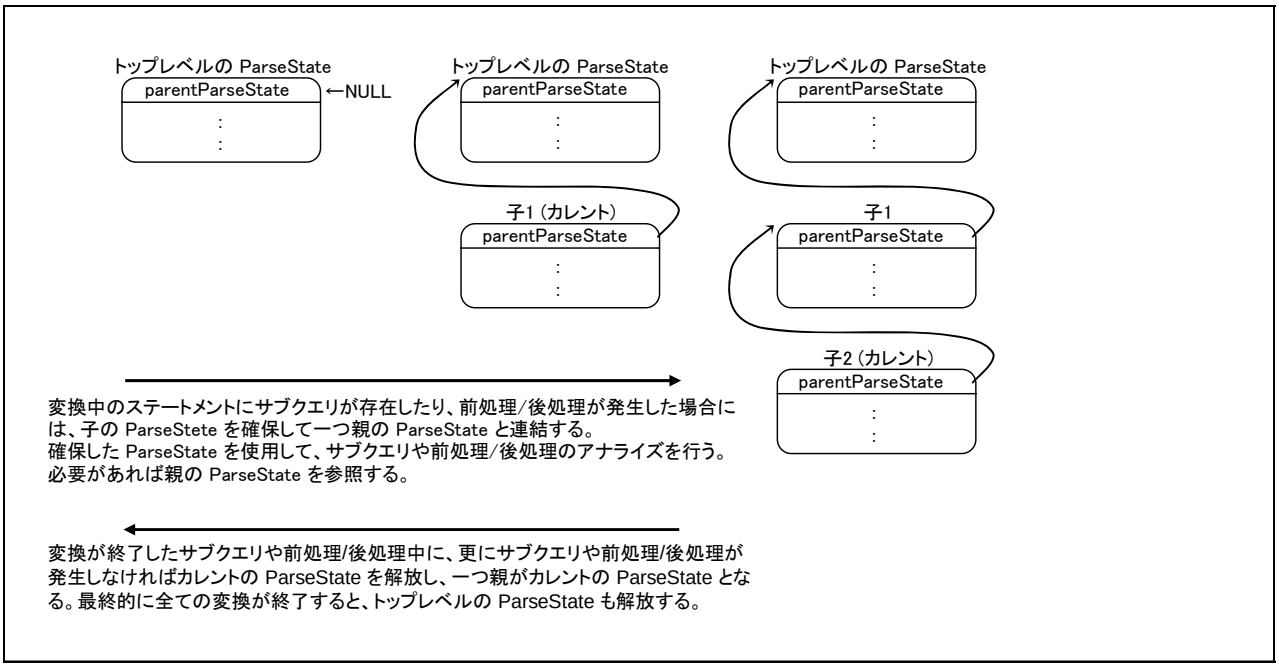

図 5-4 意味解析実行中の ParseState の階層構造

5.3. ステートメント変換処理 -- transformStmt()

transformStmt()が、ステートメント変換のエントリポイントとなる。transformStmt()では、受け取ったパーズ木のノードタグを判別し、各ステートメント別の変換処理に振り分けてクエリ木の生成、変換を行う。

最終的にクエリ木を返却するが、ステートメントが分割される場合にはクエリ木を返すと共に、未変換の前処理あるいは後処理用のステートメントノードのリストも返却する。

ノードタグに対応するステートメントと、その変換サブルーチンを以下に示す。

ノードタグ	対象となるステートメントと変換サブルーチン
T_CreateStmt	CREATE TABLE ステートメント。 transformCreateStmt()を呼び出して変換を行う。
T_IndexStmt	CREATE INDEX ステートメント。 transformIndexStmt()を呼び出して変換を行う。
T_RuleStmt	CREATE RULE ステートメント。 transformRuleStmt()を呼び出して変換を行う。
T_ViewStmt	CREATE VIEW ステートメント。 VIEW 対象のクエリは transformStmt()の再帰呼び出しで変換。
T_ExplainStmt	EXPLAIN ステートメント。 EXPLAIN 対象のクエリは transformStmt()の再帰呼び出しで変換。
T_AlterTableStmt	ALTER TABLE ステートメント。 transformAlterTableStmt ()を呼び出して変換を行う。
T_PrepareStmt	PREPARE ステートメント。 transformPrepareStmt ()を呼び出して変換を行う。

T_ExecuteStmt	EXECUTE ステートメント。 transformExecuteStmt ()を呼び出して変換を行う。
T_InsertStmt	INSERT ステートメント。 transformInsertStmt ()を呼び出して変換を行う。
T_DeleteStmt	DELETE ステートメント。 transformDeleteStmt ()を呼び出して変換を行う。
T_UpdateStmt	UPDATE ステートメント。 transformUpdateStmt ()を呼び出して変換を行う。
T_SelectStmt	SELECT ステートメント。 通常は transformSelectStmt ()を、集合演算の場合は transformSelectStmt ()を呼び出して変換を行う。
T_DeclareCursorStmt	DECLARE CURSOR ステートメント。 transformDeclareCursorStmt ()を呼び出して変換を行う。
上記以外	commandType フィールドが CMD_UTILITY の Query ノードを作成し、utilityStmt フィールドには transformStmt()が受け取ったパーズ木のステートメントノードを変換せずにそのまま格納する。

DML系のT_InsertStmt / T_DeleteStmt / T_UpdateStmt / T_SelectStmtのステートメントの場合、変換処理後のQueryノードのcommandTypeフィールドに、それぞれCMD_INSERT / CMD_DELETE / CMD_UPDATE / CMD_SELECTが格納される。それ以外のステートメントではCMD_UTILITYが格納される。またDML系と異なりCMD_UTILITYの場合には、utilityStmtフィールドにステートメントノードが格納される[16]。

以降、CREATE TALBE ステートメントの変換処理と SELECT ステートメントの変換処理を紹介する。

5.4. CREATE TABLE ステートメントの変換処理 -- transformCreateStmt ()

CREATE TABLE ステートメントノードの変換を行い、クエリ木を作成して返す。
また、クエリ木を返すと共に、ステートメントが分割される場合には、未変換の前処理あるいは後処理用のステートメントノードを作成して、そのリストを引数 extras_before, extras_after に格納して返す。
返却されるQueryノードは、基本的にcommandTypeフィールドとutilityStmtフィールドのみの書き換えとなる。レンジテーブルやターゲットリストは作成しないので、既に存在しているテーブル名でCREATE TABLEを実行しても意味解析器の時点ではエラーとならない[17]。
commandTypeフィールドにはCMD_UTILITYが、utilityStmtフィールドには変換後のCREATE

[16] ステートメントの種類によっては、transformStmt()が受け取った未変換の状態のステートメントノードがそのまま格納される場合と、変換された状態のステートメントノードが格納される場合がある。
[17] DDL系のステートメントは、基本的にレンジテーブルやターゲットリストを作成しない。クエリ木生成の時点ではリレーションや属性の有無チェックは行われず、エグゼキュータ実行時にチェックされる。ただしユーティリティ系ステートメントのexplainのように子ノード内にDML系のクエリ木ノードが含まれる場合については、意味解析中にリレーションや属性の有無エラーが発生するステートメントもある。

TABLEステートメントノードが格納される[18]。
この CREATE TABLE ステートメントノード内では、SERIAL 型や各種制約情報の変換が行われる程度で、文字列で表現されているリレーション名やデータ型の OID 変換は行なわれない。

CREATE TABLEやALTER TABLEステートメントの変換中は、以下のような作業用データ CreateStmtContext[19]を構築しながら、解析、変換処理が進行する。
カラム定義、制約情報等の変換サブルーチンでは、変換したノードを一時的に作業データに格納し、最終的に作業データを元にクエリ木を構築し直して返却する。

```
typedef struct

{
  const char        *stmtType;        /* "CREATE TABLE" か "ALTER TABLE" */
  RangeVar *relation;         /* 作成するリレーションの情報 */
  List              *inhRelations;    /* 継承関係 (INHERITS 句) */
  bool              hasoids; /* OID を持っているか */
  Oid               relOid;           /* ALTER TABLE 時の テーブルの OID */
  List              *columns;/* ColumnDef の List */
  List              *ckconstraints;   /* CHECK 制約のリスト */
  List              *fkconstraints;   /* FOREIGN KEY 制約のリスト */
  List              *ixconstraints;   /* index 作成時の制約のリスト */
  List              *blist;           /* スキーマを作る前に行う(未変換の)前処理リスト */
  List              *alist;           /* スキーマを作り出した後行う(未変換の)後処理リスト */
  IndexStmt*pkey;             /* あれば PRIMARY KEY index */
  } CreateStmtContext;
```

次にtransformCreateStmt()の処理シーケンスを図 5-5に示す。

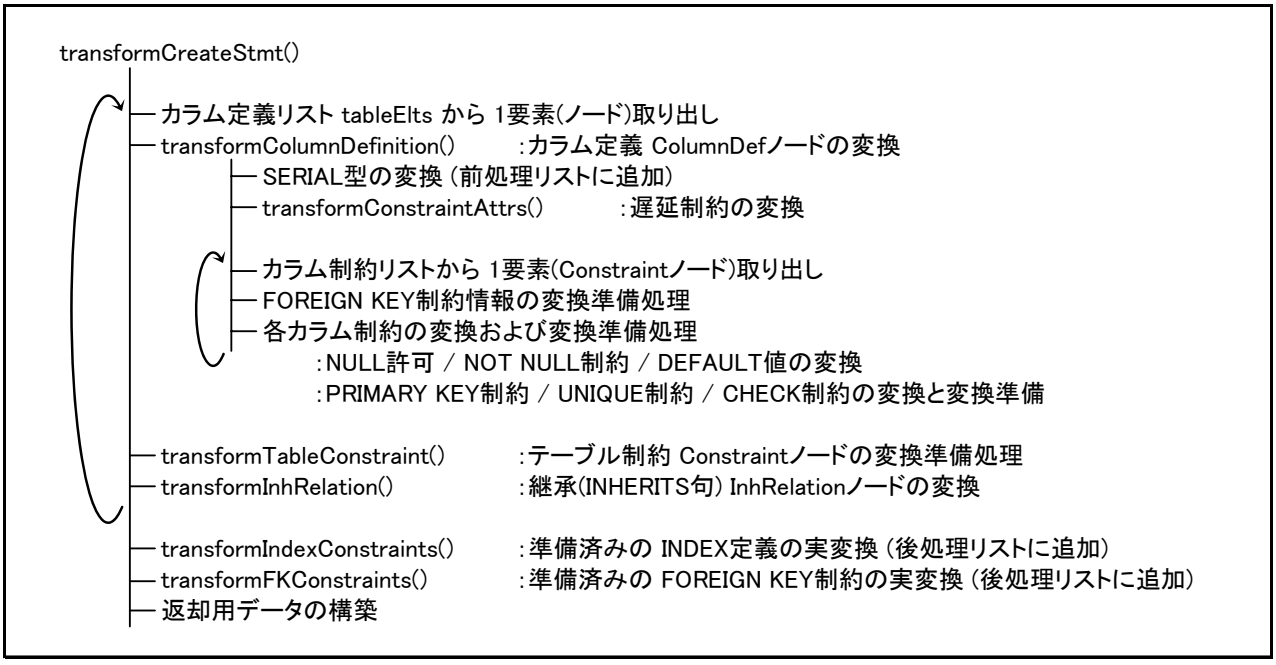

図 5-5 transformCreateStmt()の処理シーケンス

[18] 実際にはdo_parse_analyze ()でquerySourceとcanSetTagフィールドにも値がセットされる。これらの値はtransformStmt()で前処理や後処理用のステートメント分割が発生しているかどうかで異なる。
[19] ソース中のコメントではctxと略されている場合がある。

TransformCreateStmt()では、最初にステートメントノードCreateStmtのカラム定義リストtableEltsから1ノードずつ取り出して、カラム定義、テーブル制約等の変換を行い、作業データ ctx (CreateStmtContext) にセットしていく。もし取り出したノードがColoumnDefの場合には、SERIAL型の変換や各カラム制約の変換を行い、作業データ ctx に変換中の情報を保持する。
tableElts の各ノードを走査し終えたら、作業データ ctx 内をチェックして INDEX 定義が必要なカラムがあれば変換する。同様に FOREIGN KEY 制約が必要なカラムがあれば変換する。
次に作業データ ctx から変換済みの情報をステートメントノードに移行する。パーズ木の生成段階で使用されていなかった constraints フィールドは、ここで CHECK 制約のリストとしてセットされることになる。その後、返却用の Query ノードを新たに生成して commandType には CMD_UTILITY を、utilityStmt には変換後のステートメントノードをセットする。
最後に作業データ ctx 中の前処理リスト、後処理リスト、Query ノードを返却する。

CREATE TABLEステートメントの変換中にステートメント分割されるパターンは3つある[20]。
1つ目はSERIAL型の変換で、CREATE SEQUENCE ステートメントノードを前処理リストに追加する。2つ目はPRIMARY KEYやUNIQUE制約が付いている場合に、CREATE INDEX ステートメントノードを後処理リストに追加する。3つ目はFOREIGN KEY制約が付いている場合に、ALTER TABLE ステートメントノード(ADD CONSTRAINT で FOREIGN KEY 制約を追加)を後処理リストに追加している。

5.5. SELECT ステートメントの変換処理 -- transformSelectStmt()

SELECT ステートメントノードの情報をもとにクエリ木を作成して返す。
構文解析器で作成された SelectStmt は、SELECT 文の種類により若干構成が異なる。大きく分けて、次の3種類がある。

A) 一般的な SELECT 文
B) 新しいテーブル作成してSELECTの結果を投入するSELECT INTOやCREATE TABLE AS文
C) 集合を扱う SELECT 文（UNION など集合演算）

B) は into フィールドにリレーション情報が格納されている場合で、C) は SelectStmt の op フィールドが SETOP_NONE 以外の場合、A) は B) と C) 以外の場合となっている。
ここでは主に、A) 一般的な SELECT 文のクエリ木生成について説明を行う。

以下に transformSelectStmt()の処理シーケンスを示す。

[20] 5.1 節に分割されたステートメントの変換処理の説明と、図 5-2 にCREATE TABLEのステートメント分割例があるので参考にしてほしい。

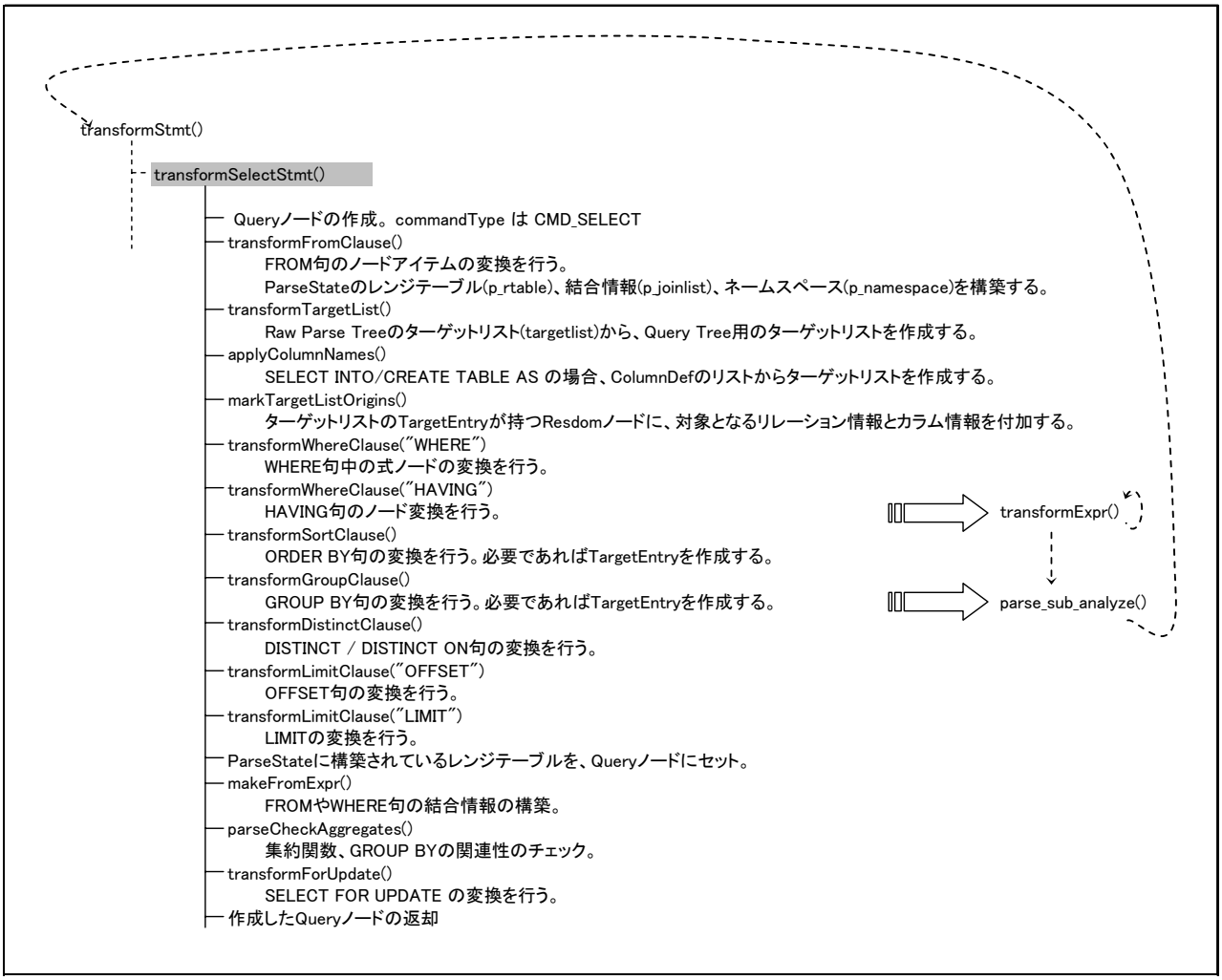

図 5-6 transformSelectStmt()の処理シーケンス

transformFromClause()では、SelectStmt の fromClause フィールドからノードを順次取り出して RangeTblEntry（RTE）ノードを作成する。全ての RTE ノードは、一時的なレンジテーブルとして ParseState の p_rtable フィールドに保持され､最終的には Query ノードの rtable に格納される｡RTE ノード作成時、通常のリレーション情報や JOIN を表現するリレーション情報のほかに、サブクエリの expression ノードが見つかった場合には parse_sub_analyze()を、関数呼び出しが見つかった場合には transformExpr()を再起的に呼び出す事によってレンジテーブル中のノード変換を進めることになる。前者は FROM 句中のサブクエリで、後者はテーブル関数呼び出しを表現している。レンジテーブルの作成中にシステムが認識できないリレーションが発見された場合は、エラーを返して処理をアボートする。またレンジテーブルの作成後は、リレーションの結合情報とネームスペース情報を ParseState の p_joinlist と p_namespaces フィールドに保持しておき、後に処理される Query ノードのターゲットリストや jointree フィールドの構築時に使用される。

transformTargetList()では、SelectStmtのtargetListフィールドからノードを順次取り出して、属性名（カラム名）がレンジテーブル中に存在するかチェックを行いながら、見つかった属性に対して ResdomノードとVarノードもしくはexpressionノードを作成する。そしてそれらのポインタを持つ

TargetEntryノードを作成する[21]。全ての属性(カラム)のTargetEntryは、QueryノードのtargetListフィールドにターゲットリストとしてリスト形式で格納する。もしレンジテーブルに対象となる属性（カラム）が見つからない場合には、エラーを返して処理をアボートする。targetListフィールド中に式（expressionノード）が存在している場合の変換は、transformExpr()を呼び出す事によって、サブクエリや関数呼び出し、演算子などの各expressionノード階層の変換を進めることになる。

transformWhereClause()では、SelectStmt の whereClause フィールドからノードを順次取り出して WHERE 句内の条件式やサブクエリ等のノード変換を行う。変換後のノードは最終的に makeFromExpr()でクエリ木ノードの jointree フィールド内に格納することになる。jointree フィールドは FromExpr ノードで構成され、リレーション結合情報を表現する RTE 参照ノードのリストを fromlist フィールドに、WHERE 句の問い合わせ条件ノードを quals フィールドに格納している。リレーションの結合情報は、通常リレーションの場合には単に RTE 参照の RangeTblRef ノードで構成されるが、JOIN の場合には JoinExpr ノードで格納され、JOIN 対象の RTE 参照や JOIN 条件を表現したノードで構成されている。これらの fromlist フィールドの情報は transformFromClause()の呼び出し時に作成された ParseState の p_joinlist をもとに構築される。quals フィールドには問い合わせの条件式ノードが格納されるが、これらの変換は transformExpr()によって再起的にノードを構築している。

他の ORDER BY 等の変換処理についても、基本的に transformExpr()呼び出しによるノード階層の構築が行われている点で同様である。

5.6. transformExpr()

transformExpr()では、演算子や関数、サブクエリなどの式ノードの変換を再起的に行う。例えば二項演算子ノードの場合は、子ノードにあたる右辺、左辺ノードを自身transformExpr()の再起呼び出しを行い変換を進める[22]。サブクエリの場合はparse_sub_analyze()を呼び出す事によってtransformStmt()が再起呼び出しされ、子ノードにあたるSelectStmtノードのQueryノード変換を進める。パーズ木で文字列表現されていたデータ型や関数、演算子などは、transformExpr()でのノード変換時に実際のOIDに変換する。演算式のデータ型合わせ（型強制）や文字列表現されている定数値の実データ型値変換なども行われるが、これらはシステムカタログ（pg_type / pg_proc / pg_operator / pg_castなど）にアクセスして変換するので、システムカタログ中のタプルに見つからない場合はエラーを返して処理をアボートする。

5.7. その他の DML 系ステートメントの変換処理

SELECT ステートメント以外の DML ステートメントの変換処理についても、レンジテーブル(rtable)、ターゲットリスト（targetList)、リレーション結合情報と問い合わせ条件（jointree）のノード変換が主であり、変換内容についても基本的に SELECT ステートメントの場合と同等である。
ただし、DELETEステートメントでターゲットリストを作成しないように、幾つか構造やフィールド

[21] SELECT * FROM …等の“*”表現の展開は、ここで行われる。
[22] expressionノード変換の再起呼び出しによるスタックオーバーフローを避けるため、ネストカウンタが DEFAULT_MAX_EXPR_DEPTH (10000 に設定) より大きくなったらtransformExpr()でエラーとしている。

の値の意味合いが異なる場合があるので、**4.1.1レンジテーブル**、**4.1.2ターゲットリスト**を参考にしてほしい。

6. 最後に

ステートメントや expression ノードの種類は多岐に渡り、またその組み合わせも膨大になるため、本ドキュメントでは幾つかに絞って説明してきた。最後にクエリ木を構成するための関数（主にレンジテーブル、ターゲットリストの作成に関わる関数）の処理概要を付録として記載したので、クエリ木解析の何らかの手助けとなれば幸いである。

付録

src/ parser/parse_clause.c

SetTargetTable()	INSERT / UPDATE / DELETE ステートメント時の RTE 作成関数。 作成した RTE の ParseState のレンジテーブル(p_rtable)内での位置を返す。 ステートメントノードの relation (RangeVar)を元にリレーションキャッシュエントリを ParseState の p_target_relation にセット。 RangeVar から RTE ノードを作成（アクセスチェックは checkForWrite）後、ParseState 内のレンジテーブル(p_rtable)に追加。 UPDATE / DELETE の場合には、ParseState の p_joinlist と p_namespace に、作成した RTE の参照表現となる RangeTblRef ノードを追加。
transformFromClause()	FROM 句のノードアイテム(通常リレーション、サブクエリ、ファンクション、JOIN ノード)の変換を行う。ParseState のレンジテーブル(p_rtable)、結合構造(p_joinlist)、ネームスペース(p_namespace)を構築する。 ステートメントノードの fromClause からノードを順次取り出して変換を行う。 RangeVar ノードの場合は RTE を作成して ParseState の p_rtable に追加。 RangeSubselect ノードの場合は parse_sub_analyze()を再帰呼び出ししてサブクエリのクエリ木を作成し、そのクエリ木からサブクエリの RTE を作成して ParseState の p_rtable に追加。 RangeFunction ノードの場合は FuncExpr ノードを作成し、その FuncExpr から RTE を作成して ParseState の p_rtable に追加。 JoinExpr ノードの場合は JOIN の左右ノードの変換を行った後、結合後の RTE を作成して ParseState の p_rtable に追加。 JoinExpr の場合は変換した JoinExpr を、RangeVar / RangeSubselect / FuncCall の場合は RTE の参照表現である RangeTblRef を作成し

	て、ParseState の p_joinlist と p_namespace の両方に追加する。 ネームスペースのリスト中で名前が衝突していないかのチェックが行われる。
transformWhereClause()	WHERE や JOIN、HAVING 等のノード変換を行う。TransformExpr()で式ノードの変換を行い、変換後のノードを返却する。 ノードの実行結果のデータ型を boolean に設定。
transformLimitClause()	LIMIT 句のノード変換を行い変換後のノードを返却する。 ノードのデータ型は int4 に設定。
transformGroupClause()	GROUP BY 句の変換を行い、GroupClause ノードのリストを返す。 必要であれば TargetEntry を作成し、引数 targetlist に追加する。
transformSortClause()	ORDER BY 句の変換を行い、SortClause ノードのリストを返す。 必要であれば TargetEntry を作成し、引数 targetlist に追加する。
transformDistinctClause()	DISTINCT / DISTINCT ON 句の変換を行う。 ターゲットリスト(引数 targetlist)から対象となる SortClause ノードを検索、あるいは作成し、SortClause ノードのリストを構築する。
addAllTargetsToSortList()	ターゲットリストからソートリストの作成を行う。 引数 targetlist の要素 TargetEntry が引数 sortlist に含まれていない場合には、SortClause ノードを作成してソートリストに追加する。 構築したソートリスト(SortClause ノードのリスト)を返す。
assignSortGroupRef()	SORT / GROUP 句で使用されている TargetEntry であれば、ターゲットリスト(引数 tlist)への参照位置(インデックス)を返す。 TargetEntry に、まだインデックスがセットされていなければセットを行い、そのインデックスをリターンする。
targetIsInSortList()	TargetEntry がソートリスト(引数 sortList)に含まれている場合には true を返す。
interpretInhOption()	引数 inhOpt を判断し、継承のオプション情報を boolean で返す。

src/ parser/ parse_relation.c

addRTEtoQuery ()	ParseState のレンジテーブル(p_rtable)内での引数 RTE の位置を探し出し、RTE への参照となる RangeTblRef ノードを作成する。 引数 addToJoinList が true ならば、作成した RangeTblRef ノードを ParseState の join アイテムリスト p_joinlist に追加する。引数 addToNameSpace が true ならば、作成した RangeTblRef ノードを ParseState のネームスペースリスト p_namespace に追加する。
RTERangeTablePosn()	ParseState のレンジテーブル(p_rtable)内での位置を返す。 引数 sublevels_up は親の ParseState を検索するネストの数。 sublevels_up が NULL ならカレントの ParseState 内の検索に留まる。レンジテーブル内に見つからなかったらエラー。

addRangeTableEntry()	リレーション情報(引数 relation)あるいは引数 alias を元に RTE ノードを作成、構築する。作成した RTE ノードを ParseState のレンジテーブル(p_rtable)に付け加えた後、RTE を返す。 もし ParseState が NULL なら RTE のみを返す。inh、inFromCl は構築する際の初期値。リレーションキャッシュ中に引数 relation に相当するリレーションが見つからなければエラー。
addRangeTableEntryForSubquery()	FROM 中のサブクエリのクエリ木(引数 subquery)を元に、RTE ノードを作成、構築する。作成した RTE ノードを ParseState のレンジテーブル(p_rtable)に付け加えた後、RTE を返す。
refnameRangeTblEntry()	ParseState 内から引数 schemaname、refname の名前にマッチする RTE を検索して返却する。 sublevels_up が NULL の場合にはカレントの ParseState のみの検索。NOT NULL の場合は、親のパーズ情報保持領域をネスト検索し、sublevels_up に検索したネスト数を格納する。
checkNameSpaceConflicts()	2 つのネームスペースノード、あるいはネームスペースのサブツリー間で名前が衝突していないか、再帰的にチェックする。衝突している場合はエラーとする。
colNameToVar()	ParseState のレンジテーブル(p_rtable)から引数指定のカラム名を検索し、Var ノードを生成して返却する。 見つからない場合は NULL を返す。もしカラム名が複数マッチした場合は、エラーとする。引数 localonly が true の場合には、親の ParseState のネスト検索は行わない。
qualifiedNameToVar()	ParseState から、引数指定のカラム "refname.colname" または "schemaname.relname.colname" のどちらかを、親 ParseState も含めて検索、そのカラムの Var ノードを生成し返却する。 引数 implicitRTEOK が true の場合、ParseState に RTE が見つかない場合に自動的にリレーション情報を付け加えて、カラムの Var ノードを生成を試みる。(旧仕様の POSTQUEL スタイル。postgresql.conf の add_missing_from パラメータ値によってはエラーとする) 見つからなかった場合は、NULL をリターン。もしリレーションやカラムが複数見つかった場合は、エラーとする。
addImplicitRTE()	旧仕様の POSTQUEL スタイルの含みをもつ RTE を作成、構築し、ParseState のレンジテーブル(p_rtable)に付け加える。
addRangeTableEntryForRelation()	リレーション OID (引数 relid)を元に、RTE ノードを作成、構築する。作成した RTE ノードを ParseState のレンジテーブル(p_rtable)に付け加えた後、RTE を返す。 もし ParseState が NULL なら RTE のみを返す。inh、inFromCl は構築する際の初期値。リレーションキャッシュ中に引数 relid に対応するリレーションが見つからなければエラーとする。

addRangeTableEntryForFunction()	ファンクション(関数)情報を元に、RTE ノードを作成、構築する。 作成した RTE ノードを ParseState のレンジテーブル(p_rtable)に付け加えた後、RTE を返す。 もし ParseState が NULL なら RTE のみを返す。inFromCl は構築する際の初期値。 関数の実行結果のデータ型チェックも実行される。
addRangeTableEntryForJoin()	JOIN の情報を元に、RTE ノードを作成、構築する。 JOIN の種別(jointype)や結合状態の Var リスト(joinaliasvars)は、この RTE に付け加えられる。もし ParseState が NULL なら RTE のみを返す。inFromCl は構築する際の初期値。
expandRTE()	RTE の情報をもとに、カラム名のリスト、Var リストを作成する。 元となる情報の参照方法は、RTE の種別 (RTE_RELATION / RTE_SUBQUERY / RTE_FUNCTION / RTE_JOIN)別に異なる。 返却用のカラム名リスト(引数 colnames)、または返却用の Var リスト(引数 colvars)が NULL の場合は、リストを作成しない。
expandRelAttrs()	RTE の情報より、TargetEntry のリスト(ターゲットリスト)を作成し返却する。カラム "*" の解析が行われ、実際のカラム情報に拡張される。
get_rte_attribute_name()	RTE から引数 attnum の位置のカラム名を取得して返却する。 get_attname()とは異なり、エイリアス名が利用できればそちらを返す。 取得できない場合は NULL を返す。
get_rte_attribute_type()	RTE から型情報(データ型の OID / 型固有情報値)を取得し、それぞれ引数 vartype / vartypmod に格納して返却する。
get_tle_by_resno()	ターゲットリスト(引数 tlist)中から、結果カラム位置(引数 resno)にマッチする TargetEntry を検索して返却する。リスト中に resno が見つからなければ NULL を返す。
attnameAttNum()	リレーション情報(引数 rd)から、引数 attname にマッチするカラム名を検索し、カラムリストの位置(カラム番号)を返却する。 引数 sysColOK が true の場合は、pg_attribute カタログも検索対象とする。
attnumAttName()	リレーション情報(引数 rd)から、カラム位置(引数 attid)のカラム情報を検索してカラム名を返却する。attid が 0 以下の場合は、システムカラム(pg_attribute カタログ)も検索対象となる。
attnumTypeId()	リレーション情報(引数 rd)から、カラム位置(引数 attid)のカラム情報を検索してデータ型 OID を返却する。attid が 0 以下の場合は、システムカラム(pg_attribute カタログ)も検索対象となる。

src/ parser/ parse_target.c

transformTargetEntry ()	通常の expression ノードを変換し、TargetEntry(ターゲットリストの要素)を作成し返却する。

	引数 node はパーズ木の expression ノードで、引数 expr は変換済みの expression ノード。未変換の場合は NULL。引数 expr が NULL あったら transformExpr()を呼び出して引数 node の変換を行う。さらに Resdom ノードを作成してから TargetEntry の生成を行う。
transformTargetList()	パーズ木 のターゲットリスト (targetlist) から、クエリ木用のターゲットリストを生成し返却する。(ResTarget のリストから TargetEntry のリストを生成) ターゲットのメタ表現 "*" の展開は、ここで行われる。 ターゲットリストの 1 要素の変換は、transformTargetEntry()を呼び出す。
markTargetListOrigins()	ターゲットリストのアイテム TargetEntry が持つ Resdom ノードに、対象となるリレーション情報とカラム情報を付加する。
updateTargetListEntry()	INSERT および UPDATE ステートメントの変換で使用される。ターゲットリストのアイテム TargetEntry(引数 tle)の Resdom に、リレーションキャッシュから取得したデータ型 OID、型固有情報を付加する。データ型の強制や、配列型の"[ ]"内の表現するサブスクリプト(引数 indirection)の変換も取り扱う。
checkInsertTargets()	INSERT のテートメントの変換で使用される。 パーズ木のターゲットリスト(引数 cols)の情報をもとにリレーション中のカラム位置のリスト(引数 attrnos)作成を行う。ターゲットリストが NULL の場合はリレーションキャッシュから削除カラムを除く全カラムの ResTarget を作成しターゲットリストを構築後、カラム位置リストを作成する。ターゲットリストとカラム位置リストを返す。

src/ parser/ parse_expr.c

parse_expr_init ()	expression ノード変換処理の初期化。expression ノード変換のネストカウンタを 0 クリアする。
transformExpr()	expression ノードのエントリポインタ。 パーズ木の expression ノード(引数 expr)の種別毎に変換処理を振り分け、変換後の expression ノードを返却する。 expression ネストカウンタのインクリメント / デクリメント、カウンタの MAX チェックはここで行われる。式の値の変換に自身 transformExpr()を呼び出ししたり、サブクエリのクエリ木作成を行う parse_sub_analyze()などが呼び出したりする事によって、再帰的に処理が実行される。データ型の強制処理と型チェックは、ここで呼び出しを行なう。
exprType()	expression ノードのデータ型の OID を返す。
exprTypmod()	expression ノードの型データ固有情報(typmod)を返す。
exprIsLengthCoerci	expression ノードがレングス(長さ)強制の対象であったら true を返す。

on()	引数 coercedTypmod には強制対象の場合の型固有情報(typmod)を格納する。

src/ parser/ parse_oper.c

LookupOperName()	演算子名と左/右辺の型をもとに pg_operator カタログから Operator オブジェクトを取得する。 Operator が見つからず、noError が true なら、InvalidOid を返す。 noError が false 場合はエラーとする。
LookupOperNameTypeNames()	演算子名と左/右辺の TypeName オブジェクトをもとに pg_operator カタログから Operator オブジェクトを取得する。 Operator が見つからず、noError が tru なら、InvalidOid を返す。 noError が false 場合はエラーとする。
equality_oper()	指定されたデータ型での "=" 演算子を表現する Operator オブジェクトを pg_operator カタログから取得する。 Operator が見つからず、noError が tru なら、InvalidOid を返す。 noError が false 場合はエラーとする。
ordering_oper()	指定されたデータ型での "<" 演算子を表現する Operator オブジェクトを pg_operator カタログから取得する。 Operator が見つからず、noError が tru なら、InvalidOid を返す。 noError が false 場合はエラーとする。
reverse_ordering_oper()	指定されたデータ型での ">" 演算子を表現する Operator オブジェクトを pg_operator カタログから取得する。 Operator が見つからず、noError が tru なら、InvalidOid を返す。 noError が false 場合はエラーとする。
equality_oper_funcid()	指定されたデータ型での "=" 演算子を表現する Operator オブジェクトを pg_operator カタログから取得し、この演算を実装している関数の OID(pg_proc.oid)を返す。 Operator が見つからない場合はエラーとする。
ordering_oper_opid()	指定されたデータ型での "<" 演算子を表現する Operator オブジェクトを pg_operator カタログから取得し、この演算を実装しているファンクションの OID(pg_proc.oid)を返す。 Operator が見つからない場合はエラーとする。
reverse_ordering_oper_opid()	指定されたデータ型での ">" 演算子を表現する Operator オブジェクトを pg_operator カタログから取得し、この演算を実装しているファンクションの OID(pg_proc.oid)を返す。 Operator が見つからない場合はエラーとする。
oprid()	指定された Operator が pg_operator カタログに存在していたら、その Operator の OID を返す。

oprfuncid()	指定された Operator が pg_operator カタログに存在していたら、この演算を実装している関数の OID(pg_proc.oid)を返す。
oper()	演算子名をもとに、pg_operator カタログから二項演算子の Operator オブジェクトを取得する。 Operator が見つからず、noError が true なら、InvalidOid を返す。 noError が false 場合はエラーとする。
compatible_oper ()	oper()とほぼ同様であるが、違いは、左右のデータ型がバイナリコンパチブルであるかのチェックが行われる。
compatible_oper_oid()	compatible_oper ()とほぼ同様であるが、違いは、pg_operator カタログのキャッシュリリースまで行う。
right_oper()	演算子名をもとに、pg_operator カタログから接尾辞に付く単項演算子の Operator オブジェクトを取得する。 Operator が見つからず、noError が true なら、InvalidOid を返す。 noError が false 場合はエラーとする。
left_oper()	演算子名をもとに、pg_operator カタログから接頭辞に付く単項演算子の Operator オブジェクトを取得する。 Operator が見つからず、noError が true なら、InvalidOid を返す。 noError が false 場合はエラーとする。
make_op()	演算子名と右辺/左辺のノードの情報をもとに、型互換性を確保しながら OpExpr (operator expression)ノードを作成する。
make_scalar_array_op()	配列の ANY() / SOME() / ALL()を表現する ScalarArrayOpExpr ノードを作成する。
make_op_expr()	Operator オブジェクトと右辺/左辺のノードの情報をもとに、型互換性を確保しながら OpExpr (operator expression)ノードを作成する。 右辺 / 左辺(あるいは両辺)のノードの型変換(型合わせ)が行われる。

src/ parser/ parse_node.c

make_parsestate ()	パーズ情報保持領域 (ParseState)の確保、初期化を行い返却する。 引数 parentParseState が NULL の場合は、トップレベルの ParseState が確保する。サブクエリや分割されたステータスの ParseState を確保する場合には親の ParseState のポインタを渡すことによって親の ParseState との紐付けを行う。
make_var ()	RTE、カラム番号(位置)をもとに、Var ノードを作成する。
transformArraySubscripts()	配列の "[ ]" 内を表現するサブスクリプトの変換。ArrayRef ノードを作成し返却する。ArrayRef ノードのデータ型は、int4 に強制する。

make_const()	grammar で返された定数の Value ノードを、本来の型、値を持つ Const ノードに変換する。 文字列系の型の場合には、型情報 UNKNOWNOID でセットされ、値が文字列の Const ノードに変換する。NULL 定数の場合には、UNKNOWNOID で型情報がセットされ、constisnull フラグがセットされた Const ノードに変換する（値はセットされない)。

src/ parser/ parse_type.c

LookupTypeName()	TypeName オブジェクトより、データ型の OID を検索して、その OID を返す。
TypeNameToString()	TypeName オブジェクトより、データ型名を作成して返却する。 通常時は catalogname.schemaname.objname や schemaname.objname 、あるいは objname の形式で名前を返す。 TypeName->pct_type が true の場合は、名前の後ろに "%TYPE" を付ける。 TypeName->arrayBounds が NOT NULL の場合は、名前の後ろに "[]" を付ける。
typenameTypeId()	TypeName オブジェクトより、データ型の OID を検索して、その OID を返す。 エラーメッセージをレポートする以外は、LookupTypeName()に相当する。
typenameType()	TypeName オブジェクトのデータ型 OID と一致した Type オブジェクトを pg_type カタログから取得する。取得に失敗した場合はエラーを上げる。
typeidIsValid()	データ型 OID が有効(システムキャッシュ上で見つかった)の場合 true
typeidType()	データ型 OID に対応した Type オブジェクト(実際は HeapTuple)を取得し返却する。
typeTypeId()	Type オブジェクトから型 OID を返す。
typeLen()	Type オブジェクトから型の内部表現内でのバイト数を返す。 (pg_type->typlen を返す) 固定長型では、typlen は型の内部表現内でのバイト数。 可変長型では typlen は負とる。 -1 は varlena 型（最初の 4 バイトにデータ長を含むもの）を意味する。 -2 は NULL 文字で終端する C 言語の文字列を示す。
typeByVal()	Type オブジェクトから、型の値が値渡しか参照渡しかの情報を返す。 (pg_type->typbyval を返す)
typeTypType()	Type オブジェクトから型の種別識別子を返す。 (pg_type->typtype を返す) 'b' は基本型

	'c' はカタログ(複合)型（テーブルの行形式） 'd' は派生(ドメイン)型 'p' は擬似型
typeTypeName()	Type オブジェクトからデータ型名を返す。 (pg_type->typname を返す)
typeTypeFlag()	Type オブジェクトから型の種別識別子を返す。 (pg_type->typtype を返す) typeTypType()と同一処理。なぜ？
typeTypeRelid()	Type オブジェクトから、関連するリレーション OID 返す。 (pg_type->typrelid を返す) カタログ(複合)型であれば、関連するテーブルを定義する pg_class のエントリを指す。基本型の場合はゼロ。
typeTypElem()	Type オブジェクトから pg_type->typele を返す。 typelem が 0 でない場合は name 型や point 型等で利用されている。
typeInfunc()	Type オブジェクトから pg_type->typinput を返す.。 入力変換ファンクション（テキスト形式）の pg_proc.oid
typeOutfunc()	Type オブジェクトから、pg_type->typoutput を返す。 出力変換ファンクション（テキスト形式）の pg_proc.oid
stringTypeDatum()	Datum 変換用関数(pg_type->typinput)を実行して、文字列を変換対象の型形式に変換する。
typeidOutfunc()	型 OID から pg_type->typoutput を返す。 出力変換ファンクション（テキスト形式）の pg_proc.oid
typeidTypeRelid()	型 OID から、関連するリレーション OID 返す。 (pg_type->typrelid を返す) カタログ(複合)型であれば、関連するテーブルを定義する pg_class のエントリを指す。基本型の場合はゼロ。
parseTypeString()	与えられた文字列をパーズして、型 OID および型修飾子に変換する。 (文字列は "int4"、あるいは "integer"、あるいは "character varying(32)" といった SQL 互換の型宣言) 引数 type_id にデータ型 OID、typemod に型固有情報を格納する。 "SELECT NULL::" ＋ 与えられた文字列のクエリからパーズ木を生成後、SelectStmt のターゲットリストから型情報を取得する。埋め込み SQL で使用されると思われるが未調査。

src/ parser/ parse_coerce.c

coerce_to_target_type ()	expression ノードのデータ型、型固有情報を、指定された型、固有情報(引数 targettype / targettypmod)に変換する。 変換後の expression ツリーをリターンする。変換が不可能の場合には NULL をリターンする。 パーザにおける一般的な型強制変換処理のエントリポイント。
coerce_type()	必要に応じて FuncExpr、RelabelType、CoerceToDomain 等のノードを作成後、それらの expression ノードのデータ型を指定の型(引数 targetTypeId)に強制し、変換後の expression ノードをリターンする。 targetTypeId が疑似(多様)型の場合は何も行わず引数 node をリターンする。
can_coerce_type()	指定されたキャストの有効範囲内で、引数 input_typeids から target_typeids の型変換が可能かどうかチェックする。 型変換可能の場合は true をリターンする。また引数 funcid には、型変換を実行するために使用する関数の OID を格納する。 ccontext はキャストの有効範囲を表す (expression での型変換か…明示的な CAST での変換か…の情報等)
coerce_to_boolean ()	引数 node のデータ型を boolean 型に型強制(キャスト)する。変換したノードを返却する。変換できなかった場合には、エラーをあげる。 constructName 引数は、エラー時のメッセージのための文字列。
coerce_to_integer()	引数 node のデータ型を integer 型に型強制(キャスト)する。
select_common_type()	expression の型リスト(typeids)の共通型を決める。(主に CASE および UNION の出力型を決定する際に利用される) データ型の OID を返却する。
coerce_to_common_type()	引数 node のデータ型を 共通の型に型強制(キャスト)する。(主に CASE および UNION 等を表現するノードの型強制で使用される)
check_generic_type_consistency()	疑似(多様)型 ANYARRAY / ANYELEMENT の型の整合性チェック。整合性が取れていれば true を返却する。
enforce_generic_type_consistency()	もし疑似(多様)型 ANYARRAY / ANYELEMENT であったら、データ型配列の一貫性のチェックを行った後、結果の型を決定して返却する。 実際のデータ型に一貫性がない場合はエラーとする。
resolve_generic_type()	引数 declared_type が疑似(多様)型の場合、実際のデータ型を決定して返却する。実際のデータ型が見つからない場合にはエラーとする。
TypeCategory()	データ型のカテゴリ分けを行う。各データ型 OID は以下の型カテゴリに分類される。 BOOLEAN_TYPE / STRING_TYPE / BITSTRING_TYPE / NUMERIC_TYPE / DATETIME_TYPE / TIMESPAN_TYPE / GEOMETRIC_TYPE / NETWORK_TYPE / UNKNOWN_TYPE / GENERIC_TYPE / USER_TYPE

IsPreferredType()	型 OID が型カテゴリとマッチしている場合は true を返す
IsBinaryCoercible()	引数 srctype の型が引数 targettype のデータ型に対してバイナリコンパチブルかどうかチェックする。バイナリコンパチブルの場合、true をリターンする。
find_coercion_pathway()	指定されたキャストの有効範囲内で、引数 sourceTypeId から引数 targetTypeId のデータ型のキャストが可能かどうかをチェックする。キャスト可能の場合は true をリターンする。また引数 funcid にはキャストを実行するために使用する関数の OID を格納する。 引数 ccontext はキャストの有効範囲を表す(expression での型変換か…明示的な CAST での変換か...の情報等)
find_typmod_coercion_function()	与えられた型 OID から型変換用関数を検索し、その関数の OID をリターンする。見つからなかった場合には InvalidOid をリターン。ただし varlena 型の場合には、F_ARRAY_LENGTH_COERCE をリターンする。また nargs には型変換関数の引数種別番号(引数の数?)を格納する。((targettype, int4)の場合は 2、(targettype, int4, bool)の場合は 3、F_ARRAY_LENGTH_COERCE の場合も 3)

<EOF>