PostgreSQL 解析資料

～ 字句解析と構文解析 ～

(株) NTT データ
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、PostgreSQL のクエリ処理で行われる字句解析と構文解析について紹介する。

一部の文字列処理のプログラムを除く、多くのプログラムでは、文字列をそのまま処理するよりも、リスト構造や木構造などのデータ構造を構築して処理する方が都合がよい。DBMS もその 1 つであり、一般的に木構造を構築してクエリのコンパイルやクエリの実行などの処理を行う。

字句解析とは、与えられたクエリの文字列から「トークン」（1 単位の文字列）を切り出す処理を行うことである。この字句解析を行うプログラムを字句解析器（scanner）と呼ぶ。そして、構文解析とは、字句解析器によって切り出されたトークンを、あらかじめ定義された文法に合わせて処理することである。この構文解析を行うプログラムを構文解析器（parser）と呼ぶ。これらの 2 つの処理は、クエリのコンパイルを行う前処理として、パーズ木と呼ばれる木構造のデータの生成を行う。

PostgreSQL では、字句解析と構文解析にそれぞれ lex と yacc（実際は、それぞれ、それらの実装の 1 つである flex と bison）を使用している。lex と yacc の知識がないために、この部分のソースコードを思うように読めない人も少なくないのではないだろうか。本ドキュメントでは、lex と yacc の知識がなくても、PostgreSQL のソースコードに使われている内容を理解できるように、lex と yacc についても説明を行っている。lex と yacc の概要をつかむ手助けになれば幸いである。しかし、lex と yacc の詳細までは説明している訳ではないので、lex と yacc の詳細については市販の書籍などを参考にしてほしい。

1.1. 対象バージョン

本ドキュメントは、PostgreSQL8.0.1 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。

2. クエリ処理の流れ

まずは、PostgreSQL のクエリ処理の概要を説明することで、字句解析と構文解析の位置付けを確認しておく。

クエリ処理は、図 2-1のようにSQL文が入力され、それをコンパイルしてプラン木を生成する。そして、作成されたプラン木をエグゼキュータと呼ばれる実行系に渡すことで、クエリ処理を実行する。クエリエグゼキュータは、プラン木に指示されたとおりに処理の実行を行うだけである。

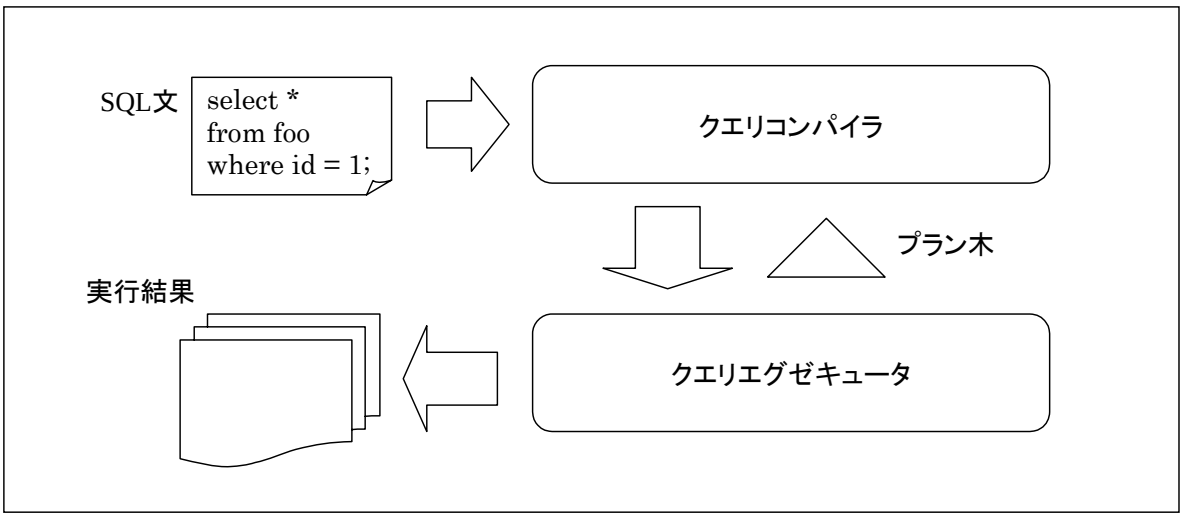

図 2-1 クエリのコンパイルと実行

クエリのコンパイルの流れを、より詳細に見てみると図 2-2のようになっている。

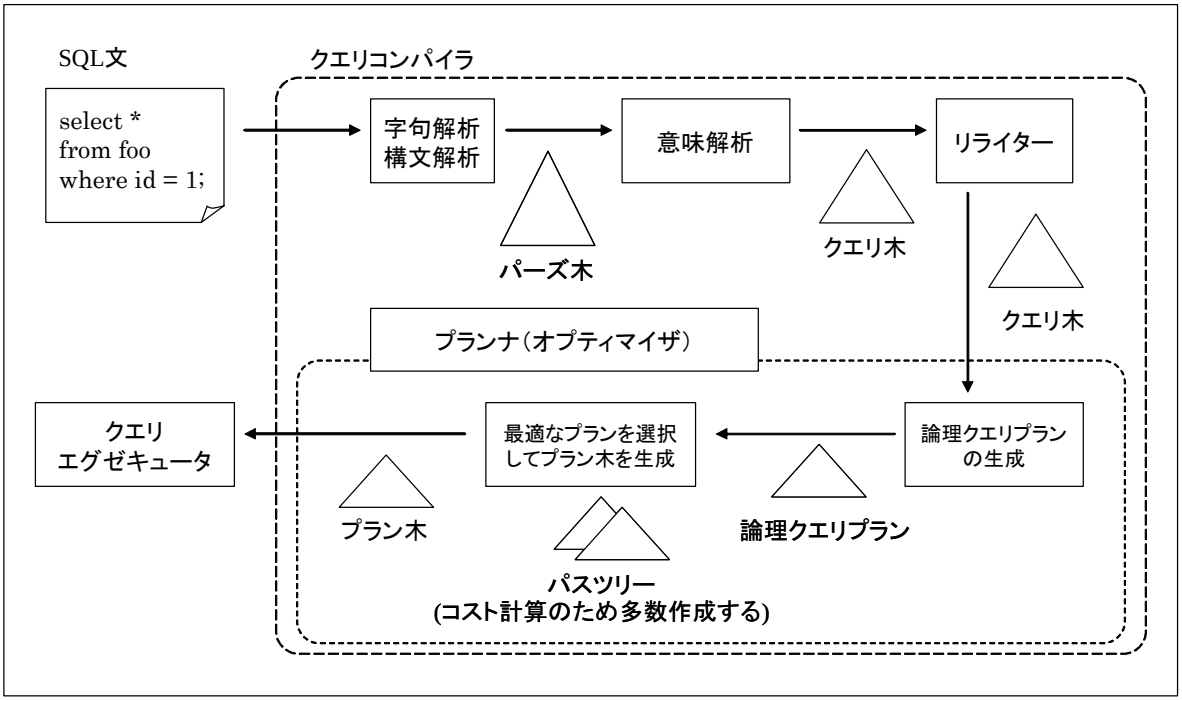

図 2-2 クエリコンパイルの流れ

まず、テキストの 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が実際にあるか確認したり、複数の SQL 文に置き換えて実行する構文の場合 SQL 文の変換を行ったりする。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。

以上が、PostgreSQL のクエリ処理の概要である。

2.1. 字句解析と構文解析の概要

ここでは、字句解析、構文解析のプログラムをもう少しだけ詳しく説明する。
字句解析では、与えられた文字列を文字列の切り出しルールに従ってパターンマッチを行い、意味のある１単位の文字列を切り出す。この切り出された文字列がトークンと呼ばれるものである。トークンは、構文解析器のソースコードの方でトークン番号が定義されている。実際の動きとしては、構文解析器が字句解析器にトークンを１つ取り出すように指示を出したら、クエリの文字列を読み進め、意味のある塊が取り出せた時点で、何番のトークンが見つかったという戻り値を返す（図 2-3）。

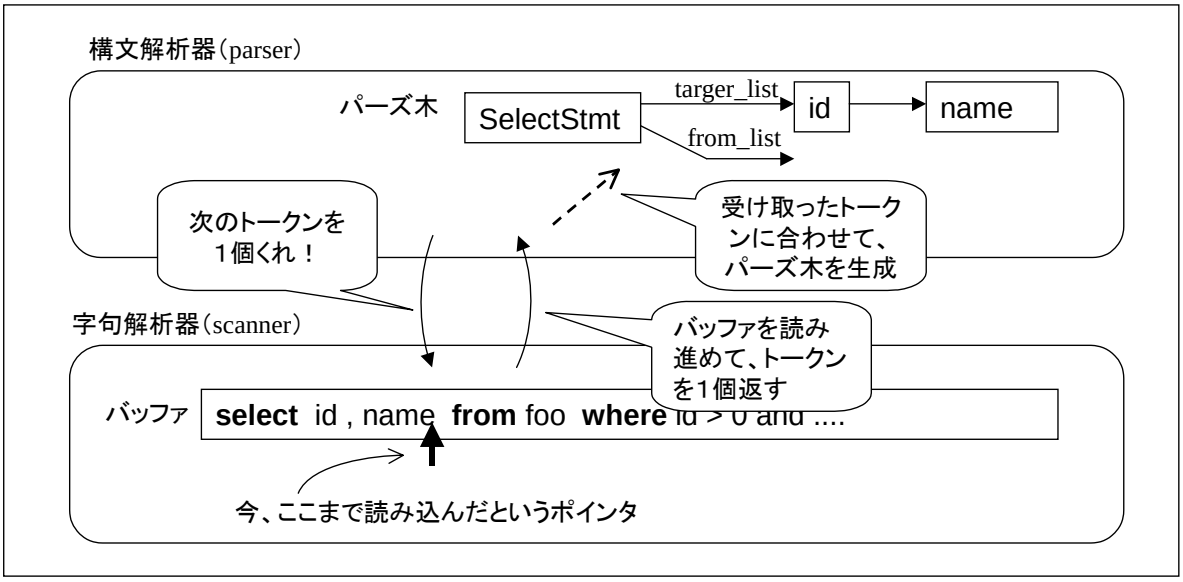

図 2-3 字句解析器と構文解析器

構文解析では、字句解析器を使ってトークンを取り出しながら、トークンの並びを構文規則の定義に従ってパターンマッチを行い、該当した構文に定義してあるアクションを実行する。アクションは、Ｃ言語風のプログラム[1]として書かれている。PostgreSQLの場合、アクションの内容は、パーズ木を生成するためのものであり、木のノードの領域をアロケートして、ノードの構造体の属性に値を設定するということを行う。その結果、パーズ木という元のSQL文に相当する木構造のデータが生成される。

[1] ほとんど C 言語と同じだが、定義部分の戻り値を$1、$2 のようにして参照することができたり、定義の戻り値と指定$$として値を設定したりするといような違いがある。

3. 字句解析

それでは、字句解析を詳しく見ていこう。字句解析器を定義してあるのは、scan.l というファイルで、lex で書かれたソースコードである。

単純な lex のソースコードと比較した場合、scan.l の特色は次のようなものが挙げられる。

1. 『SQL の予約語』と『テーブルや属性などの名称』を判別するために、キーワードを一度外部の関数に渡し、トークンを決定する。[identifier の ScanKeywordLookup()関数の説明を参照]
2. 排他的なモードを利用し、クォートで囲まれた文字列やC言語形式のコメントなどの切り出しを行う。[定義部および規則部のクォートモードの説明を参照]
3. literalbuf というバッファを使って、オリジナルの文字列に追加の文字を入れたり、変換したりして、文字列の加工を行う。具体的には、エスケープ文字の処理で、文字を追加したり減らしたりするのに使用している。

lex のソースコードは、次のように大きく分類できる。

```
%{
    /* PART 1 */
%}
    /* PART 2 */
%%
    /* PART 3 */
%%
    /* PART 4 */
```

先頭から最初の%%まで、つまり PART1 と PART2 が***定義部***と呼ばれる。次の%%まで、つまり PART3 を***規則部***と呼び、最後の部分が***ユーザ定義サブルーチン部***と呼ばれる部分である。以下、どのようなものが書かれるのかを１つずつ見ていく。

3.1. 定義部(definition section)

最初の部分であるが、%{ で始まっている部分から、%} で終わっている部分まで、つまり PART1 の部分は、C言語のコードを直接記述する。主にコメントや、インクルードファイルの定義、static な変数の定義などを書くのが一般的である。
scan.l にも、次のようにコメント、インクルードファイルの定義、変数の定義などが書かれている。

```
%{
/*-------------------------------------------------------------------------
 *
 * scan.l
 *    lexical scanner for PostgreSQL
 *
 * XXX The rules in this file must be kept in sync with psql's lexer!!!
 *
 * Portions Copyright (c) 1996-2005, PostgreSQL Global Development Group
```

```
 * Portions Copyright (c) 1994, Regents of the University of California
 *
 * IDENTIFICATION
 *    $PostgreSQL: pgsql/src/backend/parser/scan.l,v 1.119 2004/12/31 22:00:27 p
gsql Exp $
 *
 *-------------------------------------------------------------------------
 */
#include "postgres.h"

#include <ctype.h>
#include <unistd.h>

    :
    :

/* Avoid exit() on fatal scanner errors (a bit ugly -- see yy_fatal_error) */
#define fprintf(file, fmt, msg)  ereport(ERROR, (errmsg_internal("%s", msg)))

extern YYSTYPE yylval;

static int      xcdepth = 0;    /* depth of nesting in slash-star comments */
static char    *dolqstart;      /* current $foo$ quote start string */

    :
    :

/* Handles to the buffer that the lexer uses internally */
static YY_BUFFER_STATE scanbufhandle;
static char *scanbuf;

unsigned char unescape_single_char(unsigned char c);

%}
```

PART2 の部分にも、いくつかの定義がしてある。それらを順に説明していく。

3.1.1. 定義の書き方

まず、定義の書き方を説明する。PART2 の定義は、次のように書く。

```
定義の名称              定義の内容
```

定義の内容は、正規表現で記述する。定義の内容で、他の定義を引用する場合は、引用したい定義の名称を {} で囲んで使用する。具体的には、次のような感じになる。

```
whitespace      ({space}+|{comment})
```

この例では、whitespace を定義するために、space と comment の定義を引用している。

3.1.2. モード

次に、字句解析中に使用されるモードについて説明する。

字句解析では、INITIAL というのが初期モードで、かつ通常モードである。通常の SQL 構文の切り出

しは、INITIAL モードで行っている。

字句解析では、INITIAL という通常モード以外に、次のような排他的なモードが定義されている。後述の規則部では、モードを分けた上で規則が書いてある。

```
 * Exclusive states:
 *  <xb> bit string literal
 *  <xc> extended C-style comments
 *  <xd> delimited identifiers (double-quoted identifiers)
 *  <xh> hexadecimal numeric string
 *  <xq> quoted strings
 *  <xdolq> $foo$ quoted strings
 */

%x xb
%x xc
%x xd
%x xh
%x xq
%x xdolq
```

ここで定義されているモードは、特定状況下での文字列の読み出しに使われる。例えば、<xc>モードは、C 言語風のコメントであり、 /* から */ までの文字列を取り除くために使う。つまり、/* が始まった時点で、<xc>モードに入り、 */ というコメントの終わりが見つかるまで、文字を読み飛ばすのに使用する。同様に、<xq>モードの quoted strings は、’ からはじまって、’で終わるまでのデータとしての文字列を読み出すのに使われる。
他のモードも、基本的には同様で、字句解析の結果として１つの塊の文字列を取り出すのに使われる。この基本的なことを押さえておけば、規則部の中は、<>でモードの付いている部分は１つの塊として読み分ければいいことが分かる。

実際の scan.l のコードでは、%x によるモードの定義のあと、空白文字関連の定義がされていて、その後に各種のモードの定義がされている。ここでは、先にモードの定義を説明する。
各モード全てを見ても仕方がないので、今回は、クォートモード（<xq>モード）の処理を見ていこう。クォートモードに関連するものは、次のような定義がされている。これらの定義がクォートモードに関連していることは、後述の規則部を見る必要がある。規則部で<xq>{XXXXX} という規則の定義があり、この XXXXX に入るものをここに取り出している。

```
quote           '
xqstart         {quote}
xqstop          {quote}
xqdouble        {quote}{quote}
xqinside        [^¥¥']+
xqescape        [¥¥][^0-7]
xqoctesc        [¥¥][0-7]{1,3}
xqcat           {quote}{whitespace_with_newline}{quote}
```

quote は、シングルクォート１つであることが、１行目に書いてある。定義を見ると{quote} の場合、xqstart と xqstop の２つの定義があてはまる。後ほど、規則部で説明するが、実は、通常モードのと

きに xqstart が適用されて <xq> モードに入った場合だけ xqstop の定義は使われるので、これらの２つがバッティングすることはない。xqdouble であるが、シングルクォートが２つ並んだ場合に、これだと判断される。これも、xqstop とバッティングしそうであるが、より長いルールが適用されるので、シングルクォートが２つ並んだ場合は、xqdouble ということになる。
xqinside は、シングルクォートと¥以外の文字が１つ以上並んでいるということになる。
¥の後ろに 0 から 7 以外の文字がきていた場合、¥と次の文字の並びをエスケープ文字として xqescape として判断される。¥の後ろに 0 から 7 の文字が１～３個並んでいた場合、オクテット表現のエスケープ文字として判断され、xqoctesc として判断される。
最後の xqcat は、シングルクォートのあと、改行を含むホワイトスペース文字が続いて、またシングルクォートから始まった場合である。例えば、次のように複数行に分けて、文字列を書く場合に使われる。

```
insert into foo values ( 1,  'aaaaa'
                             'bbbbb'
                             'ccccc' );
```

この 'aaaaa' 'bbbbb' 'ccccc' は、後述の空白文字の定義と合わせてみると分かるのだが、 'aaaaabbbbbccccc' と等価に扱うための仕掛けである。

3.1.3. 空白文字定義

空白文字関連の定義であるが、次のようになっている。半角スペース、¥t,¥n¥r¥f は空白文字 space として扱われる。他にも水平方法のスペースと改行について定義されている。また、-- によるコメントもここで定義されている。

```
space           [ ¥t¥n¥r¥f]
horiz_space     [ ¥t¥f]
newline         [¥n¥r]
non_newline     [^¥n¥r]

comment         ("--"{non_newline}*)

whitespace      ({space}+|{comment})
```

また、同じ空白文字の定義でも、次のように、改行を含むもの、含まないもので、もう少し細かく定義している。これらは、前述の xqcat の定義などに使われている。

```
special_whitespace      ({space}+|{comment}{newline})
horiz_whitespace        ({horiz_space}|{comment})
whitespace_with_newline ({horiz_whitespace}*{newline}{special_whitespace}*)
```

3.1.4. 単一記号定義と演算子

単一記号の定義と演算子の定義は、次のようになっている。

```
self            [,()¥[¥].;¥:¥+¥-¥*¥/¥%¥^¥<¥>¥=]
op_chars        [¥~¥!¥@¥#¥^¥&¥|¥`¥?¥+¥-¥*¥/¥%¥<¥>¥=]
```

```
operator        {op_chars}+
```

self で定義されているのは、1 文字でトークンとして返す文字であり、 ,() [].:;+-*/%^<>= がそれである。次に、op_chars であるが、operator を構成する文字の集まりである。よく見ると、重複している記号があるが、最長の定義にヒットするようになっているので、定義が矛盾することはない。つまり、> は self として判断されるが、>=は、operator として判断される。
規則部を見ると、operator は、Op トークンを返すようになっている。operator は 1 文字以上の長さを持っているが、self に含まれる記号が 1 文字だけで現れた場合は、Op トークンではなく、self としてその記号をトークンとして返す。例えば、 >は 「>トークン」であるが、>= は「Op トークン」となる。

3.1.5. 数値と identifier の定義

次は、名称（identifier）と数値の定義である。説明の都合上、scan.l と定義の順序を変えてある。まず、identifier は、次のように定義されている。

```
ident_start     [A-Za-z¥200-¥377_]
ident_cont      [A-Za-z¥200-¥377_0-9¥$]
identifier      {ident_start}{ident_cont}*
```

identifier は、ident_start のうしろに ident_cont が 0 個以上続いていることになるので、1 文字目は 2 文字目以降とルールが異なるということである。定義を見てみると、A～Z と a～z のアルファベット、それと 8 進数で 0200～0377 の文字（つまり 10 進数で 128～255 の文字）とアンダスコア(_) が identifier の 1 文字目に許されている。 2 文字目以降は、それに加えて 0～9 の数値と$も許可されていることが分かる。

続いて、数値の定義である。

```
digit           [0-9]
integer         {digit}+
decimal         (({digit}*¥.{digit}+)|({digit}+¥.{digit}*))
```

これらの定義から、digit は、0 から 9 の数字 1 つで構成されていることが分かる。 integer は digit が 1 個以降続いたものであること分かり、decimal は、12.34、.34、12. のいずれかの小数点が入った形式になることが分かる。

3.2. 規則部

規則部では、定義部でマッチした塊をどのように扱うかを定義している。
通常モード（INTIAL モード）の定義は、次のような形式になっている。アクションは、C 言語に似た形式で記述する。

```
{定義の名称}  { アクション }
```

通常モードの場合は、アクションの最後は return でトークン番号（トークンの文字列定数）を返すようになっている。identifier を除き、return で返すトークン番号は、それぞれの規則の定義によって決まっているので、直接 scan.l のソースコード中に文字列定数が書かれている。例えば、{integer} のアクションの場合、return ICONST; のように、ICONST（integer の定数）というトークン番号の文字列定数を返すように書かれている。identifier だけは例外で、ScanKeywordLookup() 関数で予約語をチェックすると、それぞれの予約語に定義されているトークン番号を ScanKeywordLookup() 関数が返してくるので、その値をトークン番号として返す。

通常モードから他のモードに移る場合には、その定義のアクションの中に BEGIN(モード名);という処理を含む。

INITIAL 以外のモードでの定義の場合は、次のように先頭に<>で囲まれたモードを書いてある。

```
<モード>{定義の名称}　{ アクション }
```

モード付きの定義の場合は、ほとんどはそのモードから抜けるときに return でトークンを返す。つまり、BEGIN(INITIAL); で通常モードに戻る処理を含む定義の最後で、return でトークンの文字列定数を返す。ただし、例外として、コメントの読み飛ばしのために存在するモードでは何も返さずに、そのまま次のトークンの読み出しに移る。

3.2.1. identifier

通常の規則部では、１つの定義が return するトークンの文字列定数は常に固定なので、直接 scan.l に文字列定数が書かれている。しかし、identifier だけは例外で、予約語のチェックを行って予約語にヒットした場合は、予約語ごとのトークン番号を返す。その様子を少し詳しく見ていく。

identifier の定義は次のようになっている。

```
{identifier}    {
                    const ScanKeyword *keyword;
                    char           *ident;

                    /* Is it a keyword? */
                    keyword = ScanKeywordLookup(yytext);
                    if (keyword != NULL)
                    {
                        yylval.keyword = keyword->name;
                        return keyword->value;
                    }

                    /*
                     * No.  Convert the identifier to lower case, and truncate
                     * if necessary.
                     */
                    ident = downcase_truncate_identifier(yytext, yyleng, true);
                    yylval.str = ident;
                    return IDENT;
                }
```

切り出した文字列が、identifier であることが確定したら、この定義の中に入ってくる。identifier の定義 {identifier}では、まず、ScanKeywordLookup() 関数を使って予約語でないかどうかをチェックする。ちなみに、ScanKeywordLookup()の引数である yytext には、切り出した文字列が入っており、この部分のソースコードは、flex によって生成されている。切り出した文字列が予約語であった場合は、yylval.keyword に予約語名を代入し、スキャンの結果としてその予約語のトークン番号を返す。identifier が予約語でなかった場合、yylval.str に切り出した文字列を設定し、戻り値には IDENT というトークン番号を返す。identifier のうち、テーブル名やカラム名などユーザが定義したものは、この方法で渡される。

ScanKeywordLookup() は、parser/keywords.c の ScanKeywords[]で定義されている予約語の検索を行う。ScanKeywords[] は、次のように定義されている。

```
static const ScanKeyword ScanKeywords[] = {
    /* name, value */
    {"abort", ABORT_P},
    {"absolute", ABSOLUTE_P},
    {"access", ACCESS},
    {"action", ACTION},
    {"add", ADD},
    {"after", AFTER},
```

各行の左側のエントリがパターンマッチに使われる文字列で、例えば、”abort” にマッチすると文字列定数 ABORT_P を返す仕掛けになっている。ここの定義に書かれている文字列定数は、gram.y の %token で定義されており、gram.y をコンパイルする際にヘッダファイル parse.h として自動生成される。したがって、ここにエントリを追加する場合は、gram.y の%token の追加も必要となる。また、どのトークンに何番が割り当てられているかは、parse.h を見ればよい。

3.2.2. quote モードの規則

ここでは、定義部で見たクォートモードの続きを見てみよう。
クォートモードに関しては、次のような定義がされている。

```
{xqstart}       {
                    token_start = yytext;
                    BEGIN(xq);
                    startlit();
                }
<xq>{xqstop}    {
                    BEGIN(INITIAL);
                    yylval.str = litbufdup();
                    return SCONST;
                }
<xq>{xqdouble}  {
                    addlitchar('¥'');
                }
<xq>{xqinside}  {
                    addlit(yytext, yyleng);
                }
```

```
<xq>{xqescape}  {
                    addlitchar(unescape_single_char(yytext[1]));
                }
<xq>{xqoctesc}  {
                    unsigned char c = strtoul(yytext+1, NULL, 8);
                    addlitchar(c);
                }
<xq>{xqcat}     {
                    /* ignore */
                }
<xq>.           {
                    /* This is only needed for ¥ just before EOF */
                    addlitchar(yytext[0]);
                }
<xq><<EOF>>     { yyerror("unterminated quoted string"); }
```

{xqstart} のところで、定義部でシングルクォート１つを見つけた場合に、これにヒットするので、この規則が適用される。この中で、BEGIN(xq); が適用されると、<xq> と書かれているモードに移行する。つまりここで挙げた規則が適用されるようになる。
通常モードから他のモードに移った場合の特徴は、モードを抜けるまでトークンを返そうとしないということである。モードによっては、モードを抜けてもトークンを返さずに、次のトークンの読み出しに行くものもある。例えば、コメントの読み飛ばしのモードである<xc>モードなどである。
クォートモードの場合、<xq> モードの状態で、シングルクォート１つが見つかると {xqstop} の規則が適用されるので、BEGIN(INITIAL) が適用され、通常モードに移行する。そのときだけ、literalbuf にためておいた文字列を yyval.str にコピーして、トークンとしては SCONST であったという結論をつけて、構文解析器にトークンを返す。
{xqinside} では、通常の文字列なので、literalbuf に文字をコピーしている。{xqdouble} では、'' というシングルクォート２つ続きのエスケープシーケンスを見つけたので、シングルクォート１つに変換する処理を行っている。ここのシングルクォートに¥が付いているのは、lex のエスケープ文字としての意味である。{xqescape} では、エスケープ文字の処理を行っている。具体的には ¥t という文字列を見つけたら、タブ文字 ('¥t') に置き換えるというようなものである。 {xqoctesc} では、８進数で書かれたエスケープシーケンスを文字に置き換えている。つまり、これは、８進数の文字コードで指定されたものを文字に置き換える処理を行っているのである。
最後の<xq><<EOF>>は、<xq>モードのまま文字列が終わった場合、シングルクォートの数が合っていないので、そのままエラーにしている。

3.3. ユーザ定義サブルーチン部

ユーザ定義サブルーチン部には、次のような定義がしてある。主に、スキャン用のバッファと切り出した文字の加工に使う literalbuf の実装である。

yyerror(const char *message)	字句解析のルールに合わないようなエラーの場合の動作定義。
scanner_init(const char *str)	文字列のスキャンを行う前に呼び出す関数。ここには、字句解析の前処理を書いてある。 • スキャンバッファを確保して、引数の文字列を

	コピーする • literalbuf の初期サイズの確保と初期化を行う
scanner_finish(void)	文字列のスキャンが終わったあと呼び出される関数。scanner_init() などで行った処理のクリーンナップ処理を書いてある。 • スキャンバッファの解放を行う
addlit(char *ytext, int yleng)	バッファ literalbuf の末尾に、ytext から yleng 分の長さをコピーする。literalbuf が一杯になっていたら、バッファを拡張する。
addlitchar(unsigned char ychar)	バッファ literalbuf の末尾に、ychar を 1 文字追加する。literalbuf が一杯になっていたら、バッファを拡張する。
litbufdup(void)	バッファ literalbuf をコピーして、コピーした文字列を返す。コピーする領域は、palloc() で確保する。
unescape_single_char(unsigned char c)	エスケープ文字をエスケープしていない状態に戻す。つまり、¥n を '¥n' にしたり、¥t を'¥t'にしたりする。引数には ¥n のうちの n の部分だけを渡す。

3.3.1. literalbuf

literalbufは、文字列の切り出しの際に、規則により文字の置き換えを行ったりするような場合に使用されるバッファである。startlit()[2] を呼び出すとバッファがリセットされて空になり、addlit() でテキストを追加する。literalbufはpalloc()によって割り当てられ、各パーズサイクルの先頭で初期化される。

次のように、バッファへのポインタと、どこまで使用しているかの位置を示す変数と、アロケートしたサイズを示す変数を持っている。

```
static char    *literalbuf;     /* expandable buffer */
static int      literallen;     /* actual current length */
static int      literalalloc;   /* current allocated buffer size */
```

literalbuf は、主にエスケープ文字の置き換えのために使用される。クォートモード、つまり SQL で文字列を与えている場合、シングルクォート２つでシングルクォートをエスケープしたり、¥n や¥t などで改行文字やタブ文字をエスケープしているのを、データとして扱う際に変換したりする必要がある。この変換用のバッファとして使われるのである。

3.4. 動作例

ここでは、実際に字句解析器が動作する様子について、例を見ながら説明していく。構文解析器からyytext()を呼び出して、１トークン分のデータが要求されると、字句解析器は、入力文字列の先頭から

[2] 関数ではなく、マクロで定義されている。#define startlit()　(literalbuf[0] = '¥0', literallen = 0)

規則部の定義に従って１トークン取り出す（図 2-3)。
図 3-1の例の場合は、最初にselect まで読み出される。これは、１連の文字列がスペースで区切られているので、ここまでが１つの塊と判断されたためである。select は {identifier} として判断され、ScanKeywordLookup() が実行されて予約語に登録されているかどうかのチェックを受ける。すると、select という予約語にヒットするので、SELECT というトークンとして取り出される。そして、呼び出し元である構文解析器のyytext() の戻り値として、文字列定数SELECTの数値が、トークン番号として返される。

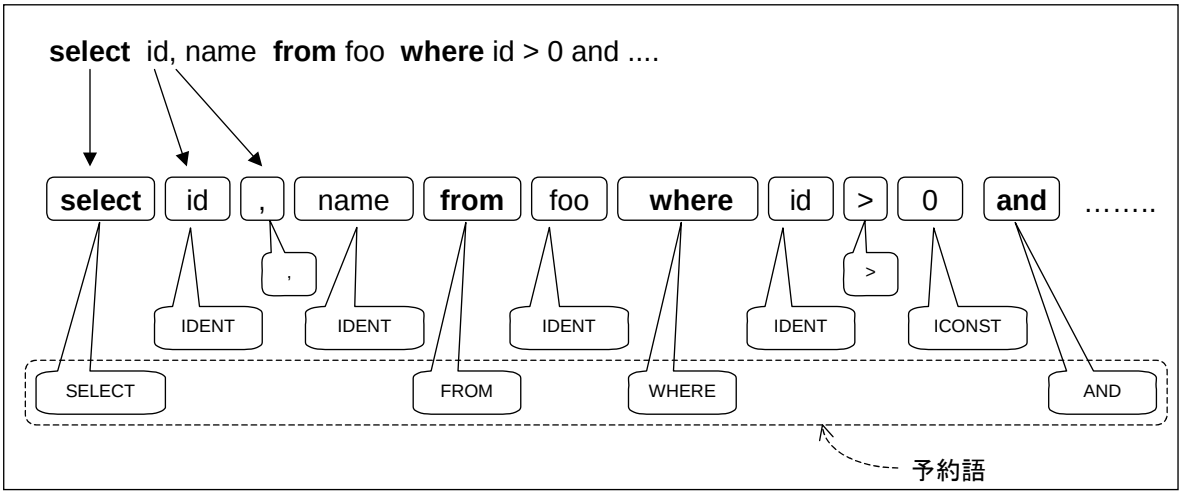

図 3-1 字句解析器によるトークンの切り出し

次に構文解析器からトークンが要求されると、同様に id が {identifier} として検出される。しかし、今度は予約語の中で見つからないので、IDENT（identifier の意味）というトークンとして構文解析器に返される。次のコンマは、記号のままトークンとして切り出される。
以下同様に、from, where and などは予約語として登録されているので、それぞれ、FROM, WHERE, AND などとして切り出され、> などは記号としてそのまま切り出される。そして、name や foo などは、IDENT として切り出される。0 については、ICONST（integer の定数）として切り出される。

3.5. 新規構文追加の際に字句解析器に追加するもの

字句解析というレベルでは、新規構文を追加する場合の多くは、新しい予約語とトークンを追加する程度の場合がほとんどである。その他に新しい文字の処理を追加したいということや、文字列切り出しのためのモードを追加したいということがない限り、それほど変更するところはない。もし、文字列切り出しのための新モードを追加したい場合は、クォートの処理などを参考にして一通りの定義と規則を追加すればよい。scan.l に修正を加えて文字列の切り出しを追加した例としては、PostgreSQL 8.0 から$をクォートの代わりにできるようにする機能が追加されたので、PostgreSQL 7.x の scan.l と見比べて見るのもいいかもしれない。

実際に一番多い、キーワードとトークンの追加であるが、トークンの文字列定数は、次の構文解析器の一部として gram.y に記述することになる。キーワードの追加は、keywords.c にキーワードとトークンの定数（gram.y に追加する文字定数）をペアで書き込む。keywords.c では、二分探索が行われるため、キーワードを正しい位置に追加する必要がある。

例えば、”ikubo” というキーワードをトークン IKUBO として追加したい場合、keywords.c には次のように書く。

```
static const ScanKeyword ScanKeywords[] = {
    /* name, value */
    {"abort", ABORT_P},
    {"absolute", ABSOLUTE_P},
       :
    {"hour", HOUR_P},
    {"ikubo", IKUBO},
    {"ilike", ILIKE},
      :
```

定数 IKUBO については、あとから gram.y の %token の定義に追加しておく必要がある。

4. 構文解析

字句解析が終わった時点で、トークンと呼ばれる、構文解析器が解釈する単位のデータが渡されてくる。実際は、字句解析を全部行ってから構文解析に移るのではなく、構文解析器が字句解析器に対してトークンを１つ渡してくれという要求を出す。すると、字句解析器はトークンを１つ切り出せるところまで処理をして構文解析器に処理を返す。いわば、ストリーム処理のような流れで処理が進んでいく。

構文解析器で行うことは、トークンの並びを、定義されている構文と比較しながらどの構文か解釈し、必要に応じてパーズ木を生成していくことである。定義されている構文にない入力を見つけた場合、エラーを報告する。もし、１つの構文の最後まで正しい入力を受け取り続けることができたなら、「構文的に正しいパーズ木」[3]が１つ生成されている。
PostgreSQL の構文解析器の場合、構文解析の処理は入力された全 SQL に対して１度で終わらせるように書かれている。そのため、複数のパーズ木がリストとして生成される作りになっている。しかし、実際に psql などから SQL を送る場合は、psql が SQL 文を１文ずつ送るようなので、大量の SQL を実行するからといって、バックエンド内に、大量のパーズ木が生成されるということはない。

4.1. gram.y 概要

gram.y を読むためには、多少なりとも yacc の知識が必要になってくる。ここでは、PostgreSQL のソースコードが追える程度に、yacc について説明しながら、gram.y を見ていく。

4.1.1. 終端記号、非終端記号

yacc の文法は、単語、つまりシンボルを用いて定義する。字句解析器が生成するシンボルをトークン、または終端記号という。構文規則の左辺で定義するシンボルを非終端記号という。これらを組み合わせて、次のように構文規則を定義していく。

[3] 「構文的に正しい」とは、構文規則上問題がなかったということである。パーズ木で指定されているテーブルやカラムが実際にあるかどうかなどの意味的にも正しいパーズ木であるかどうかのチェックは、意味解析で行われる。

```
非終端記号: 終端記号と非終端記号を組み合わせた定義　{ この左辺の非終端記号適用時のアクション }
                ;
```

全ての非終端記号は、定義の部分が終端記号だけになるまで、繰り返し定義される。

4.1.2. ソースファイルの構造

yacc のソースファイルは、次のように%%だけの行を境にして３つの部分からなる。gram.y も例外ではない。

```
  定義部
%%
  規則部
%%
  ユーザ定義サブルーチン部
```

定義部には、終端記号と非終端記号の型の定義や、演算子の結合ルールと優先度などが定義される。***規則部***には、構文規則が定義される。ここに SQL 構文が定義されている。また、それぞれの構文の規則に当てはまったときに、どのようなアクションをとるのかが記述されている。***ユーザ定義サブルーチン部***は、規則部のアクションを書くのを支援する関数が定義されている。

以降、順に説明していく。

4.2. 定義部

定義部では、非終端記号や終端記号の型や、演算子の結合の方法や優先度を定義しておく。

4.2.1. シンボルの型定義

まず、%union 宣言を用いて、シンボル値で使う型を定義する。左側は yacc のソースコードをＣ言語にコンパイルした後の型で、右側は yacc のソースコード中で使う型名である。

```
%union
{
    int             ival;
    char            chr;
    char            *str;
    const char      *keyword;
    bool            boolean;
    JoinType        jtype;
    DropBehavior    dbehavior;
    OnCommitAction  oncommit;
    ContainsOids    withoids;
    List            *list;
    Node            *node;
      :
      :
```

パーズ木の中でよく使われる node という型も、このように%union の中でC言語の Node 型のポインタとして定義されている。

4.2.2. 非終端記号の型定義

%type 宣言を使って、非終端記号の型を宣言する。非終端記号の定義の書式は次のとおりである。

```
%type <型名> 非終端記号１ 非終端記号２ ......
```

<型名> には、%union で定義した型名の１つを書く。１つの%type の行に複数の非終端記号をまとめて書くことができる。
例えば、次のように複数行に渡って node 型の終端記号を定義している部分がある。

```
%type <node>    stmt schema_stmt
        AlterDatabaseSetStmt AlterDomainStmt AlterGroupStmt AlterOwnerStmt
        AlterSeqStmt AlterTableStmt AlterUserStmt AlterUserSetStmt
            :
            :
```

当然、次のように node 型以外を使った終端記号も定義している。

```
%type <list>    alter_table_cmds alter_rel_cmds

%type <dbehavior>   opt_drop_behavior

%type <list>    createdb_opt_list copy_opt_list transaction_mode_list
%type <defelt>  createdb_opt_item copy_opt_item transaction_mode_item
```

これらは、いずれも非終端記号なので、規則部のどこかで左辺として登場する。

4.2.3. 終端記号（トークン）の型定義

%token 宣言を使って、終端記号の型を宣言する。終端記号の定義の書式も非終端記号の定義と同じで、次のとおりである。

```
%token <型名> 終端記号１ 終端記号２ ......
```

実際に定義してあるのは次のようになっている。一番多いのが、次のように keyword 型である。中には、str 型や ival 型のものもある。

```
%token <keyword> ABORT_P ABSOLUTE_P ACCESS ACTION ADD AFTER
    AGGREGATE ALL ALSO ALTER ANALYSE ANALYZE AND ANY ARRAY AS ASC
    ASSERTION ASSIGNMENT AT AUTHORIZATION

    BACKWARD BEFORE BEGIN_P BETWEEN BIGINT BINARY BIT
    BOOLEAN_P BOTH BY
```

ちなみに、gram.y で%token として定義されたトークンの文字列は、gram.y のコンパイル時に文字列

定数として C のヘッダファイル parse.h に書き出される。文字列定数に割り当てられる数字は、重複がないように連番で自動的に採番される。parse.h を見ると次のような定義が見られ、実際にどのトークンにどの番号が割り当てられているか確認できる。

```
enum yytokentype {
  ABORT_P = 258,
  ABSOLUTE_P = 259,
  ACCESS = 260,
  ACTION = 261,
  ADD = 262,
  AFTER = 263,
  AGGREGATE = 264,
```

この例は、ちょうど前述の %token の例のところである。
このようにして自動生成された parse.h を scan.l でもインクルードすることにより、文字列定数の数値を自動的に合わせることができる。

4.2.4. 結合の優先順位と右結合、左結合

%left、%right、%nonassoc で結合の順序と優先度を定義している。%left は左結合を表す。左結合とは、A－B－C があった場合に、(A－B)－C であって A－(B－C)ではないことを示す。ちなみに、後者の A－(B－C) となるものを右結合といい %right で表す。例えば、= 演算子などは右結合である。また、結合しない演算子を %nonassoc で表す。
%left、%right、%nonassoc の定義は、同じ列にあるものは同じ優先度で、上に出てくるものほど結合が弱く、下のほうに書かれているものほど結合が強いことを表している。

gram.y の定義部を見ていくと、次のような定義が見つかる。

```
/* precedence: lowest to highest */
%left       UNION EXCEPT
%left       INTERSECT
    :
    :
%nonassoc   ISNULL
%nonassoc   IS NULL_P TRUE_P FALSE_P UNKNOWN /* sets precedence for IS NULL, etc */
%left       '+' '-'
%left       '*' '/' '%'
%left       '^'
    :
    :
```

これは、UNION と EXCEPT が左結合する演算子であって、この中では一番弱い結合であることを示している。次に結合が弱い演算子は INTERSECT であり左結合であることを示している。しばらく上位の演算子にいくと、ISNULL があり結合しない演算子であることを示している。次に、IS NULL_P TRUE_P FALSE_P UNKNOWN が同レベルにあり、これらも結合しない演算子である。
以下同様に、+ と － が左結合、* と / と % も左結合、^ という順序であることが分かる。

この順序づけにより、規則部に次のような定義があった場合に、あいまいさがなくなる。

```
a_expr: a_expr '+' a_expr
                { $$ = (Node *) makeSimpleA_Expr(AEXPR_OP, "+", $1, $3); }
            | a_expr '-' a_expr
                { $$ = (Node *) makeSimpleA_Expr(AEXPR_OP, "-", $1, $3); }
            | a_expr '*' a_expr
                { $$ = (Node *) makeSimpleA_Expr(AEXPR_OP, "*", $1, $3); }
            | a_expr '/' a_expr
                { $$ = (Node *) makeSimpleA_Expr(AEXPR_OP, "/", $1, $3); }
            | a_expr '%' a_expr
                { $$ = (Node *) makeSimpleA_Expr(AEXPR_OP, "%", $1, $3); }
```

例えば、%left だけしか優先度を定義していない場合、1 + 2 * 3 は (1 + 2) * 3 と判断されてしまう。* の方が結合の度合いが強いことを定義している（下のほうに書いてある）ので、1 + (2 * 3)と判断できるのである。

4.3. 規則部

規則部では、終端記号と非終端記号を使って、次のような形式で構文規則を定義していく。

```
非終端記号: 終端記号と非終端記号を組み合わせた定義　{ この左辺の非終端記号適用時のアクション }
                ;
```

具体的には、次のようなになっている。

```
stmtblock:  stmtmulti                                   { parsetree = $1; }
        ;
```

左辺という言葉がしばしば使われるが、左辺とは、コロン（:）の左側である。そこに非終端記号のシンボルを定義して、右辺に定義とアクションを書く。この例では、stmtblock は、stmtmulti で構成されていると定義されている。そして、{ } の中にこの構文が適用されたときのアクションが書かれている。

アクションの中は、C言語と似た形式で記述する。アクションの記述では、「終端記号と非終端記号を組み合わせた定義」の中に出てきたシンボルの戻り値を、出てきた順番で$1、$2 のようにして参照できる。例えば、前述の stmtblock の例では、定義部分は stmtmulti が 1 つしかないが、1 番目である stmtmulti の結果を$1 として参照して、parsetree に代入している。

また、アクションの中では、左辺で定義した非終端記号の戻り値を $$ として設定することができる。

4.3.1. gram.y の規則部

gram.y の規則部は、前述の stmtblock から始まっており、全体を stmtblock でまとめている。これによって、全体の処理結果が parsetree という変数に代入されるという仕掛けである。

```
stmtblock:  stmtmulti                                   { parsetree = $1; }
        ;

stmtmulti:  stmtmulti ';' stmt
                { if ($3 != NULL)
```

```
                    $$ = lappend($1, $3);
                  else
                    $$ = $1;
                }
            | stmt
                    { if ($1 != NULL)
                        $$ = list_make1($1);
                      else
                        $$ = NIL;
                    }
        ;

stmt :
            AlterDatabaseSetStmt
            | AlterDomainStmt
            | AlterGroupStmt
            | AlterOwnerStmt
            | AlterSeqStmt

                :
                :
```

この規則部の定義を読んでいくと、stmtblock は stmtmulti から構成されていて、stmtmulti は stmtmulti に１つの stmt がセミコロンでつながっているか、stmt が１つであるという再帰的な定義がされている。つまり、stmtblock とは、stmt が１つ以上セミコロンで連結された stmt の集まりであることが分かる。そして、stmt の定義を見るとこれが SQL の構文の分岐点になっていることが分かり、stmt１つは、SQL 文１つであると考えていいだろう。つまり、stmt は各種の SQL 文である。

ここでは、代表で create 文（CreateStmt）を追って見よう。CreateStmt の定義は、| で結合されて２つ書かれているが、今回は１つ目だけを見ることにする。１つ目の定義は次のようになっている。

```
CreateStmt: CREATE OptTemp TABLE qualified_name '(' OptTableElementList ')'
            OptInherit OptWithOids OnCommitOption OptTableSpace
                {
                    CreateStmt *n = makeNode(CreateStmt);
                    $4->istemp = $2;
                    n->relation = $4;
                    n->tableElts = $6;
                    n->inhRelations = $8;
                    n->constraints = NIL;
                    n->hasoids = $9;
                    n->oncommit = $10;
                    n->tablespacename = $11;
                    $$ = (Node *)n;
                }
```

まず、CREATE というトークンがきて、つぎに非終端記号である OptTemp の定義を参照している。その後 TABLE というトークンが来ることになっている。OptTemp の定義は次のようになっている。

```
OptTemp:    TEMPORARY                   { $$ = TRUE; }
            | TEMP                      { $$ = TRUE; }
            | LOCAL TEMPORARY           { $$ = TRUE; }
            | LOCAL TEMP                { $$ = TRUE; }
```

```
            | GLOBAL TEMPORARY              { $$ = TRUE; }
            | GLOBAL TEMP                   { $$ = TRUE; }
            | /*EMPTY*/                     { $$ = FALSE; }
        ;
```

つまり、TEMPORALY, TEMP, LOCAL TEMPORARY, LOCAL TEMP, GLOBAL TEMPORARY, GLOBAL TEMP のいずれかのトークンの列が来ていた場合は、戻り値 $$に TRUE を設定し、何もなかった場合、つまり CREATE の次に TABLE というトークンが来ている場合は、$$に FALSE が設定されることになる。
さて、ここで CreateStmt の定義に戻るが、OptTemp の戻り値は、$2 としてアクションの中で参照できる。定義の中を見ると、$4 つまり qualified_name の定義の戻り値（たぶん、ノードが生成されて返されている）の istemp という属性に、$2 つまり OptTemp の値が入る。名前から分かるように、一時的なテーブルかどうかのフラグを設定しようとしている。
以下、qualified_name 以降は、OptTableElementList、OptInherit、OptWithOids、OnCommitOption、OptTableSpace の定義を再帰的に見ていけばよい。

次に、この CreateStmt の構文が適用された場合のアクション、すなわち { } の中の処理内容を見ていく。
まずは、makeNode(CreateStmt); で CreateStmt 用のノード構造体を作成する。makeNode はマクロになっており、include/nodes/nodes.h に次のように定義されている。

```
#define makeNode(_type_)        ((_type_ *) newNode(sizeof(_type_),T_##_type_))
```

つまり、(CreateStmt *) newNode(sizeof(CreateStmt), T_CreateStmt)) に変換される。
さらに newNode の定義を見ると次のようになっている。

```
#define newNode(size, tag) ¥
( ¥
    AssertMacro((size) >= sizeof(Node)),        /* need the tag, at least */ ¥
    newNodeMacroHolder = (Node *) palloc0fast(size), ¥
    newNodeMacroHolder->type = (tag), ¥
    newNodeMacroHolder ¥
)
```

つまり、構造体 CreateStmt 分のサイズを palloc0fast で確保し、CreateStmt->type に T_CreateStmt を代入し、全体の戻り値として palloc0fast で割り当てた領域へのポインタを返している。
まとめると、makeNode(CreateStmt)では、CreateStmt 構造体のメモリをアロケートして、その中の type に T_CreateStmt を設定しているのである。

次の、「$4->istemp = $2;」であるが、$4 は qualified_name の定義で生成されたノード（構造体）の istemp という属性に $2 の値、すなわち OptTemp の結果を代入している。

以下は、比較的分かりやすいと思う。「n->relation = $4;」のようになっているのは、makeNode で生

成した CreateStmt 構造体の relation に $4 の値、すなわち qualified_name の定義で生成されたノードへのポインタを設定しているということである。
以下同様に、tableElts には$6 の OptTableElementList、inhRelations には$8 の OptInherit、constraints には強制的に NIL、hasoids には$9 の OptWithOids、oncommit には$10 の OnCommitOption、tablespacename には$11 の OptTableSpace の定義による戻り値を設定している。

最後のアクション「$$ = (Node *)n;」では、CreateStmt の定義の結果として、makeNode で生成し、ここのアクションで各属性の定義を埋めた CreateStmt 構造体を Node 型にキャストして、戻り値として返すという定義である。つまり、CreateStmt の結果としては、CreateStmt 構造体が返される。

以上のような感じで、それぞれの構文を読んでいくことができる。

4.4. ユーザ定義サブルーチン部

最後に、ユーザ定義サブルーチン部を見てみる。基本的には、規則部のアクションを書くのを支援するものである。規則部のアクションを支援する関数は、他のファイルにも散らばっている。例えば、最もよく使われる makeNode() などは、include/nodes/nodes.h に書かれている。また、ノード作成の関数は、src/nodes/makefuncs.c などにも書かれている。

makeColumnRef()	ColumnRef ノードの作成を支援する関数。
makeTypeCast()	TypeCast ノードを作成する関数。
makeStringConst()	文字列の定数ノードを作成し、値を設定する関数。
makeIntConst()	Integer の定数ノードを作成し、値を設定する関数。
makeFloatConst()	浮動少数の定数ノードを作成し、値を設定する関数。
makeAConst()	定数ノードを作成する関数。
makeDefElem()	DefElem ノードを作成し値を設定する関数。
makeBoolAConst()	ブール値の定数ノードを作成し、値を設定する関数。
makeRowNullTest()	行ノードの属性に NULL を含んでいたか調べて、その真偽の結果をブール値のノードとして作成する関数。
makeOverlaps()	OVERLAPS 演算子を支援するために、FuncCall ノードを生成する関数。
check_qualified_name()	names のリストが全て、T_String のノードで、かつ、その値が * でないことを確認する関数。
check_func_name()	func_name としてふさわしいかチェックする関数。チェック内容は、check_qualified_name() と同じ。
extractArgTypes()	FunctionParameter ノードに、型の参照用ポインタを張る関数。
findLeftmostSelect()	set の parsetree の中から最も左の SelectStmt を探す関数。
insertSelectOptions()	SelectStmt にオプションを設定する関数。
makeSetOp()	set の演算子を作成する関数。set は、SelectStmt として作成する。

SystemFuncName()	システム組込の関数への参照を作成する関数。
SystemTypeName()	システム組込の型への参照を作成する関数。
parser_init()	１つのクエリのパーズを開始するための初期化処理関数。
exprIsNullConstant()	式に定数として NULL が入っているかどうか判断する関数。
doNegate()	負の数を扱うための関数。
doNegateFloat()	負の数を扱うための関数。

ここにある makeXXXXX() という関数は、基本的には src/nodes/makefuncs.c に移動することが可能である。src/nodes/makefuncs.c にも同じ役割の関数群が定義されている。

これらの関数の定義が終わった後に、scan.c をインクルードしている。これによって、字句解析器のコードを結合している。

```
#include "scan.c"
```

4.5. 実行例

今度は、具体的な実行例を挙げて、構文解析が行われる様子について説明を行う。
次のような SQL 文を実行したときに作成されるパーズ木を考えてみる。

```
create table foo ( id int );
insert into foo values ( 1 );
select * from foo;
```

これまでの話を総合すると、複数のSQLステートメントは、1 つのリストにまとめられ、parsetree_list[4]は、図 4-1のようになると想像するだろう。

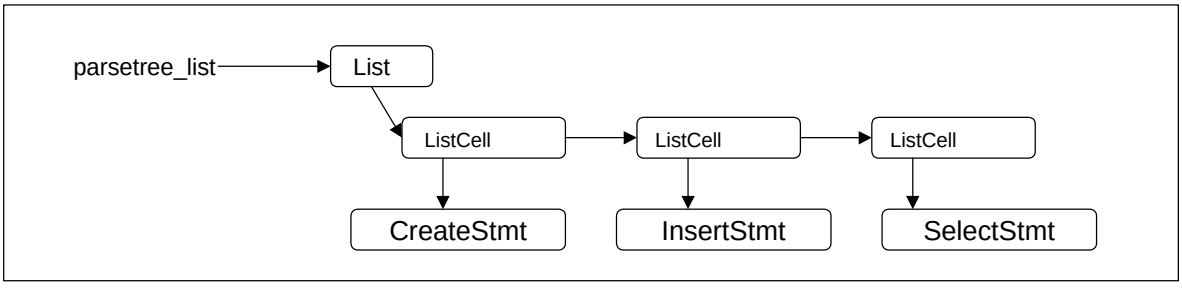

図 4-1 パーズ木のリスト（予想）

しかし、psql から前述のSQLを実行した場合、SQL文が 1 つずつパーザに渡されるため、実際は、図 4-2のように 1 つずつのリストが 3 回処理されることになる。

[4] gram.y では、非終端記号 stmtblock の定義で、parsetree という変数にパーズ木のリストを渡している。これは、構文解析器の呼び出し元で、すぐに parsetree_list という変数に代入されるので、ここでは、parsetree_list という変数名を使用している。したがって、これまでの parsetree と同じと考えてよい。

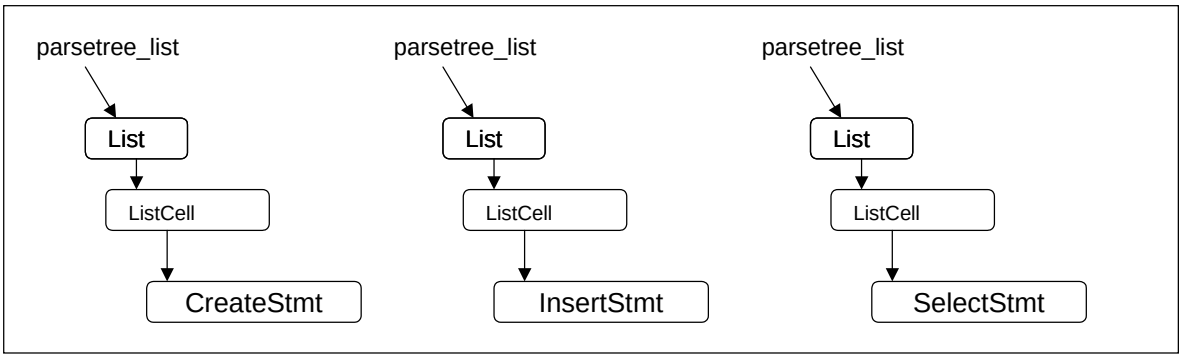

図 4-2 パーズ木のリスト（実際）

これは、１行に３つの SQL 文を書いたり、begin ～ end を使ってトランザクションとして囲んだりしても同じである。
このように、SQL 文を分割して処理する理由は、大量の SQL 文を実行するとパーザの処理で大量のメモリと処理時間を必要とするためであると考えられる。

SQL 文は、個別に処理されることが分かったので、ここでは、select 文の処理に注目してみよう。
まず、字句解析器の結果は次のようなトークンの列を返してくることになる。

```
SELECT * FROM IDENT ;
```

このトークンの列を見て、最初にSELECT がきているので SelectStmt であると判断し、SelectStmt の処理としてSelectStmt ノードが生成される。その中で、今回は targetList に * 、fromClauseのリストに foo が設定される（図 4-3）。

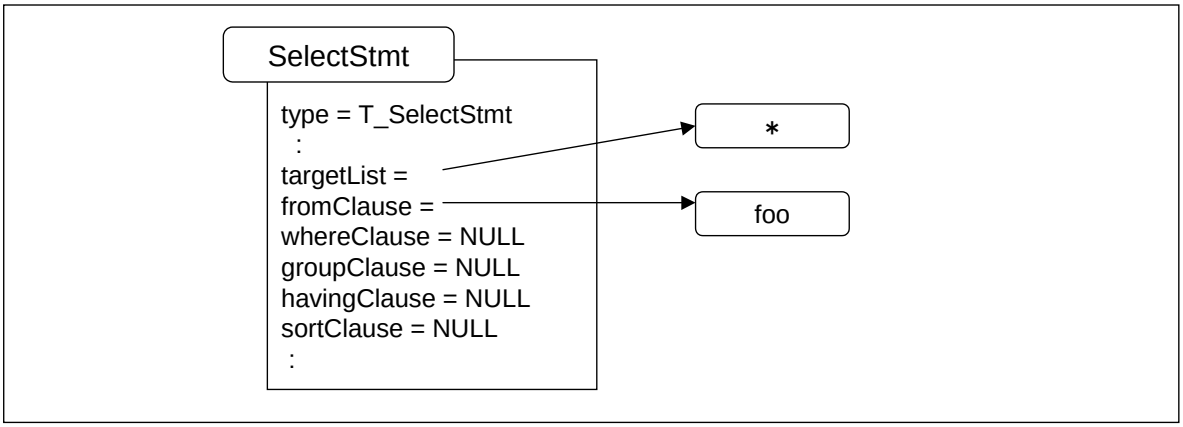

図 4-3 SelectStmt の構造

foo という FROM 句が設定されるところをもう少し詳しく見てみよう。SelectStmt は、少々複雑な複数の定義があるが、今回ヒットするのは、simple_select の最初の定義である。

```
simple_select:
            SELECT opt_distinct target_list
            into_clause from_clause where_clause
            group_clause having_clause
```

```
                {
                    SelectStmt *n = makeNode(SelectStmt);
                    n->distinctClause = $2;
                    n->targetList = $3;
                    n->into = $4;
                    n->intoColNames = NIL;
                    n->intoHasOids = DEFAULT_OIDS;
                    n->fromClause = $5;
                    n->whereClause = $6;
                    n->groupClause = $7;
                    n->havingClause = $8;
                    $$ = (Node *)n;
                }
```

ここで、from_clause の定義を見ると次のようになっており、FROM トークンの次に from 句のリストが来ることになっている。

```
from_clause:
            FROM from_list                          { $$ = $2; }
            | /*EMPTY*/                             { $$ = NIL; }
        ;
```

以下、関係のあるところの定義だけ取り出してくると次のようになっている。実際は、この順番に定義されているわけではないので注意してほしい。

```
from_list:
            table_ref                               { $$ = list_make1($1); }
            | from_list ',' table_ref               { $$ = lappend($1, $3); }
        ;

table_ref:  relation_expr
                {
                    $$ = (Node *) $1;
                }

relation_expr:
            qualified_name
                {
                    /* default inheritance */
                    $$ = $1;
                    $$->inhOpt = INH_DEFAULT;
                    $$->alias = NULL;
                }

qualified_name:
            relation_name
                {
                    $$ = makeNode(RangeVar);
                    $$->catalogname = NULL;
                    $$->schemaname = NULL;
                    $$->relname = $1;
                }

relation_name:
            SpecialRuleRelation                     { $$ = $1; }
            | ColId                                 { $$ = $1; }
        ;
```

```
ColId:      IDENT                                   { $$ = $1; }
```

最後の ColId のところで、トークンが IDENT であった場合、その値を戻り値として返すようになっている。
IDENT は、終端記号であり、文字列のトークンであることが定義されている。

```
%token <str>    IDENT FCONST SCONST BCONST XCONST Op
```

IDENT の戻り値としての文字列は、字句解析器から渡されてくる“foo”という文字列である。
ここでようやく終端記号だけの定義にたどり着いたので、from 句の処理は、呼び出し元に戻り始める。
呼び出し元に戻っていく過程で、from 句を示す非終端記号 from_clause の結果は、RangeVar というノードのリストとして返される。

5. おわりに

以上で、PostgreSQL のクエリ処理における字句解析と構文解析について説明した。全ての構文を追った訳ではないが、このドキュメントにより、字句解析器や構文解析器のソースコードを読む手助けになれば幸いである。
＜ＥＯＦ＞