Database Languages and their Compilers

Prof. Dr. Torsten Grust

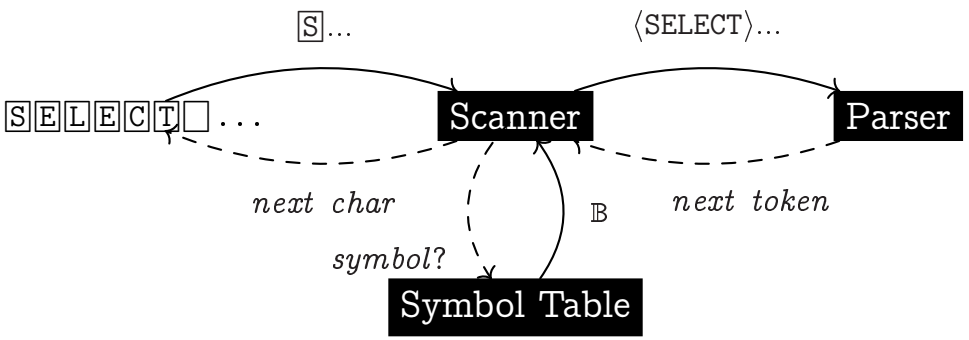
Database Systems Research Group
U Tübingen

Winter 2010

1 Query Parsing

1.1 The Scanner (Lexer)

- On the most basic level, any query input is nothing but a **stream of characters**, including **whitespace** (e.g., ␣, ↰, ⇥);
- a **scanner** (or **lexer**) drops whitespace, and maps sequences of characters to **tokens** (⟶ returns, ⇢ calls):



- The parser drives the whole scanning and parsing process.

- Example tokenization:

```
SELECT  *          ≡  [S][E][L][E][C][T][ ][*][↰]
  FROM  R             [F][R][O][M][ ][R][↰]
 WHERE  A = 42        [W][H][E][R][E][ ][A][ ][=][ ][4][2][↰]
```

$\xrightarrow{\text{scan}}$

$\langle \texttt{SELECT} \rangle\ \langle \texttt{STAR} \rangle$
$\langle \texttt{FROM} \rangle\ \langle \texttt{ID}, \texttt{"R"} \rangle$
$\langle \texttt{WHERE} \rangle\ \langle \texttt{ID}, \texttt{"A"} \rangle\ \langle \texttt{EQ} \rangle\ \langle \texttt{NUM}, 42 \rangle$

- The SQL scanner will generate a number of different tokens which can be classified as follows (with $b :: \mathbb{B}, n :: \mathbb{N}, s :: \mathbb{S}$)[3]:

Reserved words	Symbols	Constants	Identifiers
$\langle \texttt{SELECT} \rangle, \langle \texttt{SUM} \rangle, \langle \texttt{AND} \rangle$	$\langle \texttt{STAR} \rangle, \langle \texttt{DOT} \rangle, \langle \texttt{EQ} \rangle$	$\langle \texttt{BOOL}, b \rangle, \langle \texttt{NUM}, n \rangle, \langle \texttt{STRING}, s \rangle$	$\langle \texttt{ID}, s \rangle$

- **Reserved words** (and symbols) make the parsing of a language a lot simpler:
  - SELECT FROM, SELECT FROM WHERE, SELECT, FROM WHERE SELECT = FROM
  - The scanner uses a **symbol table** to recognize reserved words in the input:

$$\mathit{symbol?}\ (\texttt{"SELECT"}) \dashrightarrow \mathit{true} \qquad \mathit{symbol?}\ (\texttt{"Mail"}) \dashrightarrow \mathit{false}\ .$$

[3] b, n, and s are known as **semantic token values**.

A Hand-Crafted Scanner

```
   WHILE true DO
   |  c ← nextchar();
1  |  IF c ∈ {□, ↰, ⇥} THEN
   |  |  ;
2  |  IF c ∈ {0 ... 9} THEN
   |  |  n ← val(c);
   |  |  c ← nextchar();
   |  |  WHILE c ∈ {0 ... 9} DO
   |  |  |  n ← 10 * n + val(c);
   |  |  |  c ← nextchar();
   |  |  pushback(c);
   |  |  RETURN ⟨NUM, n⟩;
3  |  IF c ∈ {.,,, =, <, >, +, -, *, /} THEN
   |  |  s ← c;
   |  |  IF c ∈ {<, >} THEN
   |  |  |  c ← nextchar();
   |  |  |  IF c ∈ {=, >} THEN
   |  |  |  |  s ← s@c;
   |  |  |  ELSE
   |  |  |  |  pushback(c);
   |  |  RETURN lookup(s);
4  |  IF c ∈ {a ... z, A ... Z, _} THEN
   |  |  s ← c;
   |  |  c ← nextchar;
   |  |  WHILE c ∈ {a ... z, A ... Z, _, 0 ... 9} DO
   |  |  |  s ← s@c;
   |  |  |  c ← nextchar();
   |  |  pushback(c);
   |  |  IF symbol?(s) THEN
   |  |  |  RETURN lookup(s);
   |  |  ELSE
   |  |  |  RETURN ⟨ID, s⟩;
```

✎ Some remarks on
$nextchar(), pushback(c), lookup(s)$

- This scanner implementation makes use of recurring patterns of code:
  - Check for **presence** of a specific character $\boxed{c}$ (or a set of characters):

```
IF c ∈ {␣, ↰, ↦} THEN
    ...;
```

  - Check for a **sequence** of specific characters:

```
IF c ∈ {<, >} THEN
    c ← nextchar();
    IF c ∈ {=, >} THEN
        ...;
```

  - Check for **repetitions** of the same character (or set of characters):

```
WHILE c ∈ {a ... z, A ... Z, _, 0 ... 9} DO
    ...;
    c ← nextchar();
```

1.1.1 Regular Expressions

A Regular Expression Pattern (Reg.Ex.)
. . . matches a well-defined set of character strings. Any reg.ex. is built by using the building blocks listed below.

▶ Let Σ be an ordered set of characters (e.g., the ASCII code set). A **reg.ex. over** Σ takes one of the following forms:

① $\boxed{c}$ — $\boxed{c} \in \Sigma$, matches the character $\boxed{c}$ only;

② $[\boxed{c}-\boxed{c'}]$ — $\boxed{c}, \boxed{c'} \in \Sigma$, matches any character in $\{\boxed{c} \ldots \boxed{c'}\}$;

③ $[\hat{}\boxed{c}-\boxed{c'}]$ — $\boxed{c}, \boxed{c'} \in \Sigma$, matches any character *not* in $\{\boxed{c} \ldots \boxed{c'}\}$;

④ $re\ re'$ — re, re' reg.ex., matches any **sequence** of strings matched by re and re';

⑤ re^* — re reg.ex., matches 0 or more **repetitions** of strings matched by re;

⑥ $re \mid re'$ — re, re' reg.ex., matches any string matched by re **or** re';

⑦ ε — matches the empty string "" only;

⑧ (re) — re reg.ex., matches exactly the strings matched by re.

- Some handy abbreviations are in common use, e.g.:

$$\begin{array}{rcl} re^+ & \equiv & re\ re^* \\ re\,? & \equiv & re \mid \varepsilon \\ . & \equiv & [\boxed{\texttt{c}}-\boxed{\texttt{c}'}] \quad \text{with } \{\boxed{\texttt{c}} \ldots \boxed{\texttt{c}'}\} = \Sigma \\ [\boxed{\texttt{c}}-\boxed{\texttt{c}'}\boxed{\texttt{d}}-\boxed{\texttt{d}'}] & \equiv & [\boxed{\texttt{c}}-\boxed{\texttt{c}'}] \mid [\boxed{\texttt{d}}-\boxed{\texttt{d}'}] \end{array}$$

Example 1.1

- A crude reg.ex. to match e-mail addresses like `"foo@bar.org"` (with $alpha \equiv [\boxed{\texttt{a}}-\boxed{\texttt{z}}\boxed{\texttt{A}}-\boxed{\texttt{Z}}\boxed{\_}]$, $alphanum \equiv alpha \mid [\boxed{\texttt{0}}-\boxed{\texttt{9}}]$):

$$alphanum^+\ \boxed{\texttt{@}}\ (alphanum^+\ \boxed{.})^+\ alpha^+$$

- Matching a floating point literal like `"123.45E-10"` (with $digit \equiv [\boxed{\texttt{0}}-\boxed{\texttt{9}}]$):

$$(\boxed{+} \mid \boxed{-})?\ ((digit^*\ \boxed{.}\ digit^+) \mid (digit^+\ (\boxed{.}\ digit^*)?))\ (\boxed{\texttt{E}}\ (\boxed{+} \mid \boxed{-})?\ digit^+)?$$

Matching a C-style comment

Our SQL processor will accept (and ignore) C-style /*...*/ comments. How could a suitable reg.ex. look like?

1.1.2 Regular Expressions for SQL

- The set of reg.ex. needed to identify SQL tokens is actually quite simple:

Reserved words (and symbols) SELECT GROUP ws^+ BY <=

Constants

Numbers	$[0-9]^+$
Booleans	(TRUE) | (FALSE)
Strings	' $[\hat{}']^*$ '

Identifiers $[a-zA-Z\_]\ [a-zA-Z\_0-9]^*$

Negative number literals?

Why don't we match negative number constants like -42 here?

Rob Pike's Regular Expression Matcher

```
1   /* match: search for regexp anywhere in the text */
2   int match (char *regexp, char *text)
3   {
4     if (regexp[0] == '^')
5       return matchhere(regexp + 1, text);
6     do {  /* must look even if string is empty */
7       if (matchhere(regexp, text))
8         return 1;
9     } while (*text++ != '\0');
10    return 0;
11  }
12
13  /* matchhere: search for regexp at the beginning of text */
14  int matchhere(char *regexp, char *text)
15  {
16    if (regexp[0] == '\0')
17      return 1;
18    if (regexp[1] == '*')
19      return matchstar(regexp[0], regexp + 2, text);
20    if (regexp[0] == '$' && regexp[1] == '\0')
21      return *text == '\0';
22    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
23      return matchhere(regexp + 1, text + 1);
24    return 0;
25  }
```

```
1
2   /* matchstar: leftmost longest search for c*regexp */
3   int matchstar(int c, char *regexp, char *text)
4   {
5     char *t;
6
7     for (t = text; *t != '\0' && (*t == c || c == '.'); t++)
8       ;
9     do { /* * matches zero or more */
10      if (matchhere(regexp, t))
11        return 1;
12    } while (t-- > text);
13    return 0;
14  }
```

- Obviously, reg.ex. provide a flexible tool to concisely specify scanners. A possible *nexttoken*() routine:

Build a table with entries $re_i \mapsto \langle t_i, n_i \rangle, \quad i = 1 \ldots$;

```
WHILE nextchar() ≠ EOF DO
    pushback();
    Call nextchar() repeatedly and match the input against all re_i;
    IF input s matches re_j THEN
        RETURN ⟨t_j, n_j⟩;
    ELSE
        Call some default action;
```

  - **N.B.** *nexttoken*() repeatedly scans the input for the **longest** match it can find (consider the SQL reg.exs. `[O][R]` and `[O][R][D][E][R]` ws^+ `[B][Y]`).
  - In general, the value of n_j will depend on the actual text matched against re_j (consider entry $[\texttt{[0]}-\texttt{[9]}]^+ \mapsto \langle \texttt{NUM}, n \rangle$):

```
IF input s matches re_j THEN
    RETURN ⟨t_j, n_j(s)⟩
ELSE
    ...;
```

1.1.3 `lex` – A Scanner Generator

- The **scanner generator** `lex` implements this approach:
  - **input**: a table of entries $re_i \mapsto \langle t_i, n_i \rangle$;
  - **output**: a C routine `yylex()` that implements a scanner for the specified language.

- A `lex` input file consists of a series of definitions and the actual table entries (separated by %%).

- In an action, the actual matched input s is available in the variable `char *yytext`, it's length in `int yyleng`.

- A semantic token value has to be assigned to the global variable `int yylval`. The entry $re_j \mapsto \langle t_j, n_j(s) \rangle$, is thus implemented by:

```
re_j        { yylval = n_j(yytext); return t_j; }
```

```
%{
#include <tokens.h>
%}

digit                                   [0-9]
ws                                      [ \t\n]
identifier                              [a-zA-Z_][a-zA-Z_0-9]*
other                                   .

%%

"'"([^'\\\n]|\\.|\\\n)*"'"              { *(yytext+yyleng-1) = '\0';
                                          yylval = (int)yytext+1; return STRING; }
{digit}+                                yylval = atoi(yytext);    return NUM;
"TRUE" | "FALSE"                        yylval = *yytext == 'T';  return BOOL;
"SELECT"                                                          return SELECT;
"GROUP"{ws}+"BY"                                                  return GROUPBY;
{identifier}                            yylval = (int)yytext;     return ID;
"/*"([^/]|[^*]"/")*"*"+"/" | {ws}+      ;
{other}                                                           return *yytext;
```

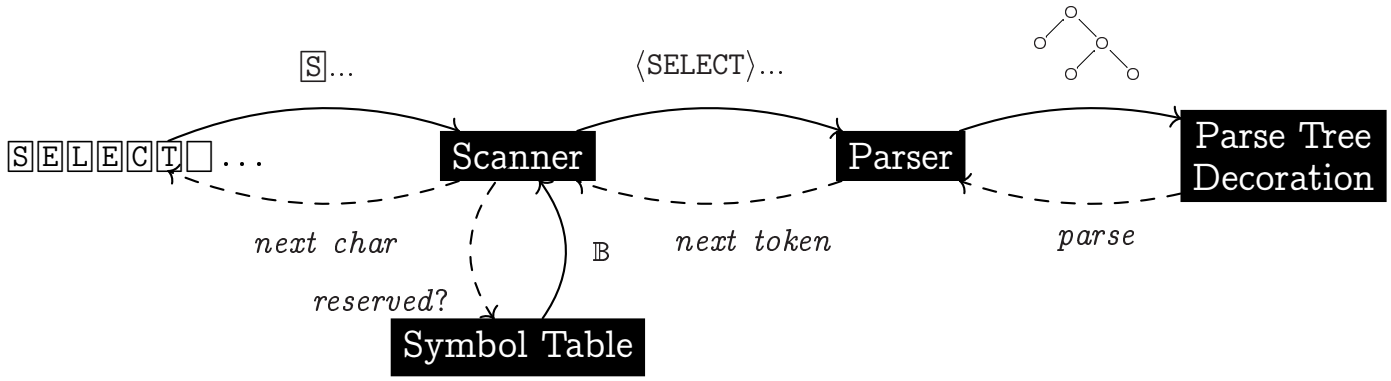
✎ `lex` Input File

A few details about the `lex` input snippet printed on the previous slide and a live demo of the same.

1.2 Syntax Analysis (Parsing)

- We will use **context-free grammars**, a set of rules precisely describing the valid queries, to implement a **parser** for SQL.

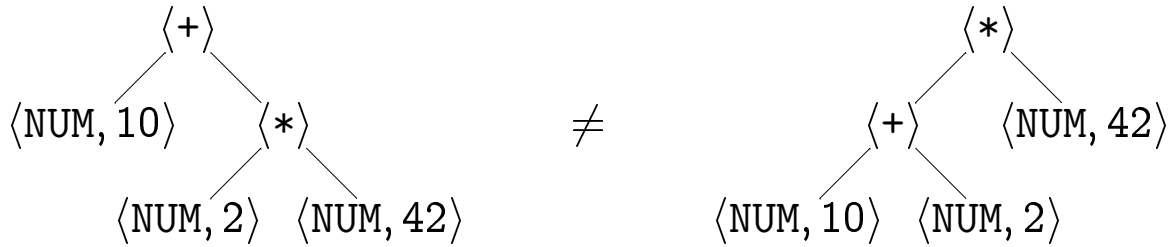
- The parser will
  - implement the **syntax check**;
  - generate a **parse tree** representing the input **query structure**:



- Such parsers are easy to specify, extensible, and may be derived from the associated grammar in a fully automated fashion.

- Although the scanner emits a **linear token sequence**, the parser has to reveal the implicit **tree-shaped structure**:[4]
  - $10 + 2 * 42 \xrightarrow{\text{scan}} \langle \text{NUM}, 10 \rangle \langle + \rangle \langle \text{NUM}, 2 \rangle \langle * \rangle \langle \text{NUM}, 42 \rangle$

    The well-known language of mathematics imposes the following structure on the above token sequence:



> ✎ **Dangling Tokens**
> Will there be a problem in determining the tree-shape for the following query?
>
> ```
> SELECT  *
>   FROM  R, SELECT  *
>            FROM  S
>  WHERE  A = 42
> ```

[4] The input language's **semantics** determines the tree shape for a token sequence.

1.2.1 Context-Free Grammars

- A **list of production rules** facilitates query syntax checking as well as parse tree construction.
  - **Production rule** i:
    $$S_{i_0} \quad \rightarrow \quad S_{i_1}\ S_{i_2}\ \ldots\ S_{i_n}$$
  - **Symbols** S_{i_j} $(j \geq 0)$ are either **terminal** (i.e. tokens) or **nonterminal**;
  - the left-hand symbol S_{i_0} is always nonterminal;
  - S_{i_0} may appear on the right-hand side of its production rule (**recursion**).

- A list of production rules forms a **context-free grammar** iff
  - there is at least one rule of the form $S_i \rightarrow \ldots$ for all nonterminal symbols S_i occurring on the right-hand side of a rule;
  - one nonterminal symbol is marked as the **start symbol** (conventionally, S_{0_0} is the start symbol).

- The following context-free grammar describes the syntax of (simplified) SQL SELECT–FROM blocks:

$$
\begin{array}{rcl}
\mathit{sf} & \to & \langle \texttt{SELECT} \rangle\ \mathit{proj\_list}\ \langle \texttt{FROM} \rangle\ \mathit{from\_list} \\
\\
\mathit{proj\_list} & \to & \mathit{expr} \\
\mathit{proj\_list} & \to & \mathit{proj\_list}\ \langle , \rangle\ \mathit{proj\_list} \\
\\
\mathit{from\_list} & \to & \langle \texttt{ID}, \_ \rangle \\
\mathit{from\_list} & \to & \mathit{from\_list}\ \langle , \rangle\ \mathit{from\_list} \\
\\
\mathit{expr} & \to & \langle \texttt{ID}, \_ \rangle \\
\mathit{expr} & \to & \langle \texttt{NUM}, \_ \rangle \\
\mathit{expr} & \to & \mathit{expr}\ \langle + \rangle\ \mathit{expr} \\
\mathit{expr} & \to & \mathit{expr}\ \langle * \rangle\ \mathit{expr} \\
\mathit{expr} & \to & \langle (\rangle\ \mathit{expr}\ \langle) \rangle
\end{array}
$$

- $\mathit{sf}, \mathit{proj\_list}, \mathit{from\_list}, \mathit{expr}$ are the nonterminal symbols of this grammar; sf is the start symbol.

N.B. The grammar is indifferent to the semantic token values ($\langle \texttt{NUM}, \_ \rangle$ vs. $\langle \texttt{NUM}, n \rangle$).

1.2.2 Derivations and Parse Trees

- Using this context-free grammar, we can perform a **syntax check** for a given query (token sequence) q as follows:

  $d \leftarrow S_{0_0}$;
  WHILE $\exists$ *nonterminal symbol* S_{i_0} *in* d DO
  
  Choose any rule $S_{i_0} \rightarrow S_{i_1}\ S_{i_2}\ \ldots\ S_{i_n}$;
  Replace S_{i_0} *by* $S_{i_1}\ S_{i_2}\ \ldots\ S_{i_n}$ *in* d;

  RETURN $d = q$;

  - If this procedure returns *true*, q is a syntactically valid SELECT–FROM block;
  - if we list d's values before each replacement step we obtain a **derivation** for q.

> **Choice of Rule**
> For one grammar and one query q there may be many (or none) possible derivations.

- One possible derivation for the query below is shown here (nonterminal symbol chosen for next replacement):

```
SELECT  A, A + 1
  FROM  R, S
```

sf

↳ ⟨SELECT⟩ *proj_list* ⟨FROM⟩ *from_list*

↳ ⟨SELECT⟩ *proj_list* ⟨FROM⟩ *from_list* ⟨,⟩ *from_list*

↳ ⟨SELECT⟩ *proj_list* ⟨FROM⟩ *from_list* ⟨,⟩ ⟨ID, "S"⟩

↳ ⟨SELECT⟩ *proj_list* ⟨FROM⟩ ⟨ID, "R"⟩ ⟨,⟩ ⟨ID, "S"⟩

↳ ⟨SELECT⟩ *proj_list* ⟨,⟩ *proj_list* ⟨FROM⟩ ⟨ID, "R"⟩ ⟨,⟩ ⟨ID, "S"⟩

↳ ⟨SELECT⟩ *proj_list* ⟨,⟩ *expr* ⟨FROM⟩ ⟨ID, "R"⟩ ⟨,⟩ ⟨ID, "S"⟩

↳ ⟨SELECT⟩ *proj_list* ⟨,⟩ *expr* ⟨+⟩ *expr* ⟨FROM⟩ ⟨ID, "R"⟩ ⟨,⟩ ⟨ID, "S"⟩

↳ ⟨SELECT⟩ *proj_list* ⟨,⟩ *expr* ⟨+⟩ ⟨NUM, 1⟩ ⟨FROM⟩ ⟨ID, "R"⟩ ⟨,⟩ ⟨ID, "S"⟩

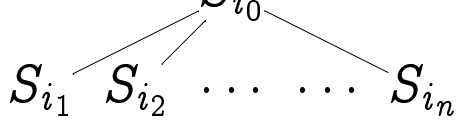
↳ ⟨SELECT⟩ *proj_list* ⟨,⟩ ⟨ID, "A"⟩ ⟨+⟩ ⟨NUM, 1⟩ ⟨FROM⟩ ⟨ID, "R"⟩ ⟨,⟩ ⟨ID, "S"⟩

↳ ⟨SELECT⟩ *expr* ⟨,⟩ ⟨ID, "A"⟩ ⟨+⟩ ⟨NUM, 1⟩ ⟨FROM⟩ ⟨ID, "R"⟩ ⟨,⟩ ⟨ID, "S"⟩

↳ ⟨SELECT⟩ ⟨ID, "A"⟩ ⟨,⟩ ⟨ID, "A"⟩ ⟨+⟩ ⟨NUM, 1⟩ ⟨FROM⟩ ⟨ID, "R"⟩ ⟨,⟩ ⟨ID, "S"⟩

- We can modify our syntax check procedure for a query q to automatically produce a **parse tree** as a side-effect during derivation:

$d \leftarrow S_{0_0}$;
$t \leftarrow S_{0_0}$;
WHILE $\exists$ *nonterminal symbol* S_{i_0} *in* d DO
  Choose any rule $S_{i_0} \rightarrow S_{i_1}\ S_{i_2}\ \ldots\ S_{i_n}$;
  Replace S_{i_0} *by* $S_{i_1}\ S_{i_2}\ \ldots\ S_{i_n}$ *in* d;
  Replace S_{i_0} *by*

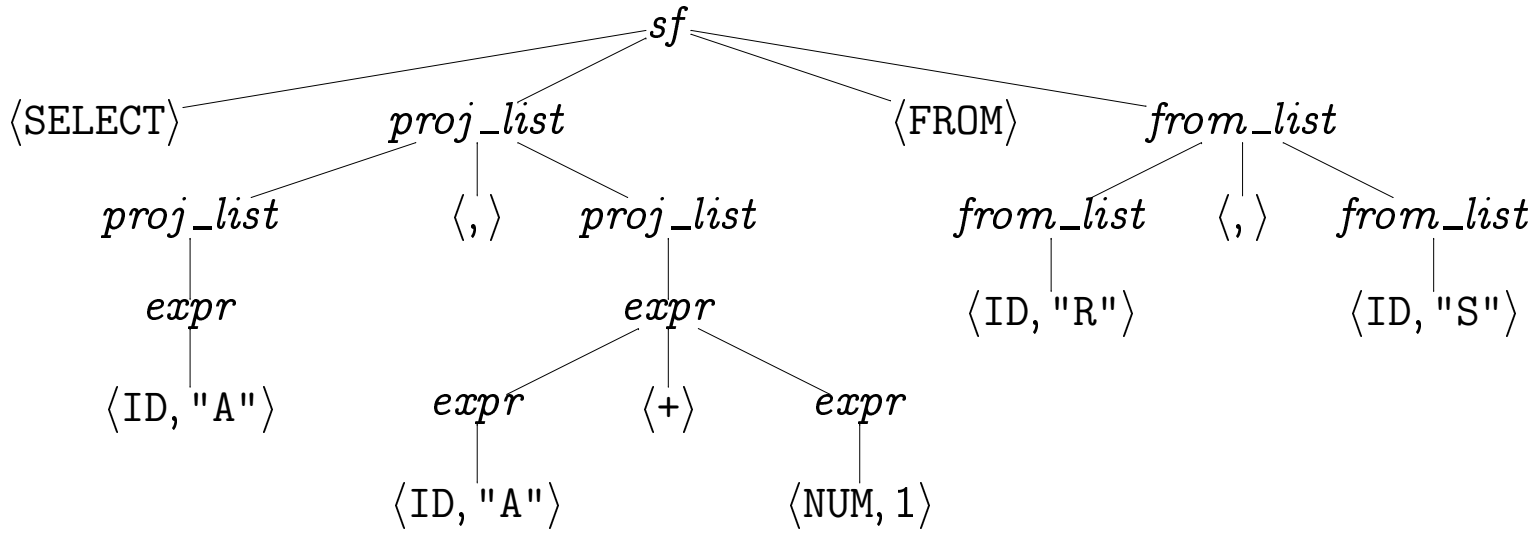


  in t;

RETURN $(d = q, t)$;

- These parse trees provide exactly the tree-shaped query structure we were aiming for in this section.

- The derivation for q shown earlier corresponds to the following parse tree:

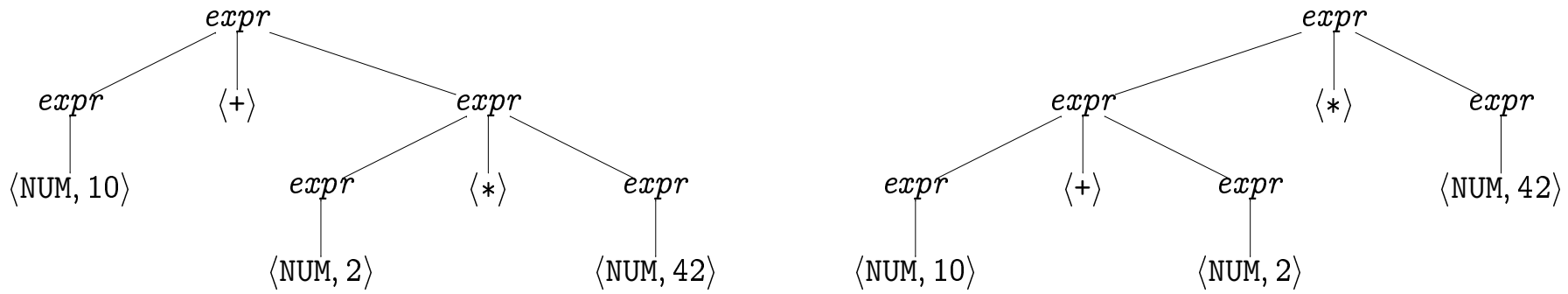


- For a successfull syntax check, a **pre-order traversal** of the parse tree's leaves[5] equals q.

[5] Also known as the tree's **front**.

1.2.3 Ambiguous Grammars

- There are grammars for which we can derive **different** parse trees for one input;
- obviously, this is a **bad thing**: parsers use parse trees to derive meaning (see the beginning of our discussion of parsers).
  - Given the simplified `SELECT–FROM` block grammar (using *expr* as the start symbol), we can derive two parse trees for the input `10 + 2 * 42`:



  - SQL's notion of the usual **operator precedence** is not reflected in the grammar.

✎ Comma-Separated Lists

Have a closer look at the (sub-)parse tree (start symbol *from_list*) for the relation list `R, S, T` in the SQL query

```
SELECT  A
  FROM  R, S, T .
```

Any ambiguity here? If there is, do such list-like constructs deserve special attention?

- We can encode operator precedence in a context-free grammar.

  ① **Identify the precedence levels** and the operators in each level (operators on higher levels bind tighter):

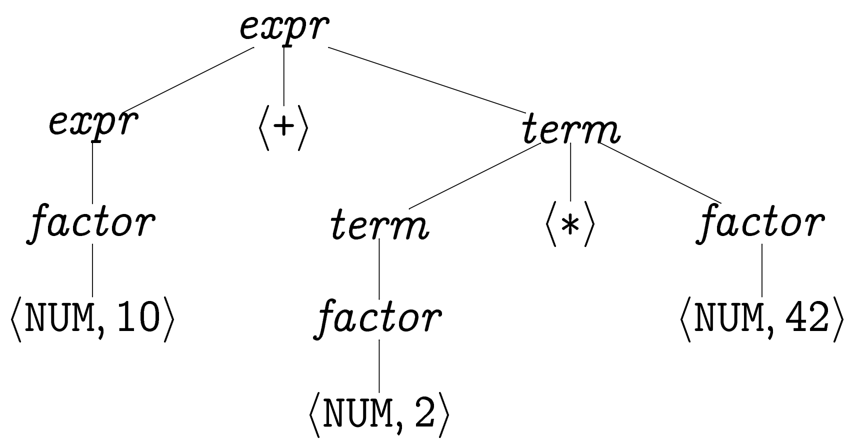
level	operator(s)
⋮	⋮
6	`* /` ...
5	`+ -` ...
⋮	⋮

  ② Introduce new **separate production rules for each precedence level**:

$$
\begin{array}{lcll}
\mathit{expr} & \to & \mathit{expr}\ \langle \texttt{+} \rangle\ \mathit{term} & \left.\vphantom{\begin{array}{l}a\\a\\a\end{array}}\right\} \text{level 5} \\
\mathit{expr} & \to & \mathit{expr}\ \langle \texttt{-} \rangle\ \mathit{term} & \\
\mathit{expr} & \to & \mathit{term} & \\
 & & & \\
\mathit{term} & \to & \mathit{term}\ \langle \texttt{*} \rangle\ \mathit{factor} & \left.\vphantom{\begin{array}{l}a\\a\\a\end{array}}\right\} \text{level 6} \\
\mathit{term} & \to & \mathit{term}\ \langle \texttt{/} \rangle\ \mathit{factor} & \\
\mathit{term} & \to & \mathit{factor} & \\
 & & & \\
\mathit{factor} & \to & \langle \texttt{ID}, \_ \rangle & \\
\mathit{factor} & \to & \langle \texttt{NUM}, \_ \rangle & \\
\mathit{factor} & \to & \langle \texttt{(} \rangle\ \mathit{expr}\ \langle \texttt{)} \rangle &
\end{array}
$$

- Leftmost derivation and parse tree for the expression `10 + 2 * 42` (start symbol *expr*):

$$
\begin{aligned}
& \mathit{expr} \\
\curvearrowright\ & \mathit{expr}\ \langle + \rangle\ \mathit{term} \\
\curvearrowright\ & \mathit{factor}\ \langle + \rangle\ \mathit{term} \\
\curvearrowright\ & \langle \mathtt{NUM}, 10 \rangle\ \langle + \rangle\ \mathit{term} \\
\curvearrowright\ & \langle \mathtt{NUM}, 10 \rangle\ \langle + \rangle\ \mathit{term}\ \langle * \rangle\ \mathit{factor} \\
\curvearrowright\ & \langle \mathtt{NUM}, 10 \rangle\ \langle + \rangle\ \mathit{factor}\ \langle * \rangle\ \mathit{factor} \\
\curvearrowright\ & \langle \mathtt{NUM}, 10 \rangle\ \langle + \rangle\ \langle \mathtt{NUM}, 2 \rangle\ \langle * \rangle\ \mathit{factor} \\
\curvearrowright\ & \langle \mathtt{NUM}, 10 \rangle\ \langle + \rangle\ \langle \mathtt{NUM}, 2 \rangle\ \langle * \rangle\ \langle \mathtt{NUM}, 42 \rangle
\end{aligned}
$$



> ✎ **Precedence captured?**
> How does this grammar transformation capture operator precedence? How about associativity of operators (remember `10 - 20 - 30`)?

1.2.4 Predictive Parsing and Recursive Descent

- Once we have constructed a context-free grammar for a language, we can **derive a parser program** from it using a simple strategy.

  In essence, the **production rules for nonterminal symbol** S_{i_0} are turned into a **single procedure** $S_{i_0}()$. $S_{i_0}()$ is recursive whenever the production rules for S are:

$$S_{i_0} \quad \rightarrow \quad S_{i_1}\ S_{i_2}\ \ldots\ S_{i_n}$$

① S_{i_j} nonterminal: call procedure $S_{i_j}()$;

② S_{i_j} terminal $\langle t \rangle$: call $eat(\langle t \rangle)$.

$eat(\langle t \rangle)$:

```
IF lookahead = ⟨t⟩ THEN
  | lookahead ← nexttoken();
ELSE
  | error();
```

③ Parsing starts by executing: $lookahead \leftarrow nexttoken();\ \ S_{0_0}();$

▶ Translate the following excerpt of the SQL grammar into a parser program:

$$
\begin{array}{rcl}
 & \vdots & \\
\mathit{quantifier} & \rightarrow & \langle \texttt{FORALL} \rangle\ \mathit{var}\ \langle \texttt{IN} \rangle\ \mathit{coll}\ \langle : \rangle\ \mathit{condition} \\
\mathit{quantifier} & \rightarrow & \langle \texttt{EXISTS} \rangle\ \mathit{var}\ \langle \texttt{IN} \rangle\ \mathit{coll}\ \langle : \rangle\ \mathit{condition} \\
 & & \\
\mathit{var} & \rightarrow & \langle \texttt{ID}, \_ \rangle \\
 & \vdots &
\end{array}
$$

Procedure *quantifier*():

```
SWITCH lookahead DO
  CASE ⟨FORALL⟩
    eat(⟨FORALL⟩); var(); eat(⟨IN⟩); coll(); eat(⟨:⟩); condition();
  CASE ⟨EXISTS⟩
    eat(⟨EXISTS⟩); var(); eat(⟨IN⟩); coll(); eat(⟨:⟩); condition();
  OTHERWISE
    error();
```

Procedure *var*():

```
eat(⟨ID, _⟩);
```

✎ Excerpt: `lex`-based predictive recursive descent SQL parser

A recursive descent parser (using a `lex`-built scanner) for a subset of SQL clauses:

$$
\begin{array}{rcl}
\mathit{quantifier} & \rightarrow & \langle \texttt{FORALL} \rangle\ \mathit{var}\ \langle \texttt{IN} \rangle\ \mathit{coll}\ \langle \texttt{:} \rangle\ \mathit{condition} \\
\mathit{quantifier} & \rightarrow & \langle \texttt{EXISTS} \rangle\ \mathit{var}\ \langle \texttt{IN} \rangle\ \mathit{coll}\ \langle \texttt{:} \rangle\ \mathit{condition} \\
 & & \\
\mathit{coll} & \rightarrow & \mathit{table\_ref} \\
 & & \\
\mathit{condition} & \rightarrow & \mathit{var}\ \mathit{rel}\ \mathit{var} \\
 & & \\
\mathit{rel} & \rightarrow & \langle \texttt{=} \rangle \\
\mathit{rel} & \rightarrow & \langle \texttt{<>} \rangle \\
\mathit{rel} & \rightarrow & \langle \texttt{<=} \rangle \\
\mathit{rel} & \rightarrow & \langle \texttt{>=} \rangle \\
\mathit{rel} & \rightarrow & \langle \texttt{<} \rangle \\
\mathit{rel} & \rightarrow & \langle \texttt{>} \rangle \\
 & & \\
\mathit{var} & \rightarrow & \langle \texttt{ID}, \_ \rangle \\
\mathit{table\_ref} & \rightarrow & \langle \texttt{ID}, \_ \rangle
\end{array}
$$

- Constructing a recursive descent parser for a given grammar seems simple enough. Let us add the VALUES SQL clause:

$$
\begin{array}{rcl}
\mathit{coll} & \rightarrow & \mathit{table\_ref} \\
\mathit{coll} & \rightarrow & \mathit{coll\_const} \\
 & & \\
\mathit{coll\_const} & \rightarrow & \langle \texttt{VALUES} \rangle\ \mathit{coll\_elems} \\
 & & \\
\mathit{coll\_elems} & \rightarrow & \mathit{const} \\
\mathit{coll\_elems} & \rightarrow & \mathit{coll\_elems}\ \langle , \rangle\ \mathit{const}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{const} & \rightarrow & \langle \texttt{NUM} \rangle
\end{array}
$$

Procedure *coll_elems*():

```
SWITCH lookahead DO
  CASE ⟨NUM⟩      ①
    const();
  CASE ⟨NUM⟩      ②
    coll_elems(); eat(⟨,⟩); const();
  OTHERWISE
    error();
```

- Procedure *coll_elems*() faces a dilemma: given only **one lookahead token** (marked below) how can it determine the proper case?

Input	Case
`...VALUES 42`	①
`...VALUES 2, 10, 42`	②

A single lookahead token must suffice

Predictive parsing will only work if knowledge of the **next terminal symbol** (the lookahead) is sufficient to determine which production rule to use next.

- Given (the right-hand side of) a production rule $S_{i_0} \to S_{i_1} \ldots S_{i_n}$, $FIRST(S_{i_1} \ldots S_{i_n})$ is the **set of all tokens that can occur as the first token in any token sequence derivable from** $S_{i_1} \ldots S_{i_n}$.
  - Given this grammar

$$\begin{array}{rcl} \mathit{term} & \to & \mathit{term}\ \langle * \rangle\ \mathit{factor} \\ \mathit{term} & \to & \mathit{term}\ \langle / \rangle\ \mathit{factor} \\ \mathit{term} & \to & \mathit{factor} \end{array}$$

$$\begin{array}{rcl} \mathit{factor} & \to & \langle \texttt{ID}, \_ \rangle \\ \mathit{factor} & \to & \langle \texttt{NUM}, \_ \rangle \\ \mathit{factor} & \to & \langle (\rangle\ \mathit{term}\ \langle) \rangle \end{array}$$

  we have

$$FIRST(\mathit{term}\ \langle * \rangle\ \mathit{factor}) = \{\langle \texttt{ID}, \_ \rangle, \langle \texttt{NUM}, \_ \rangle, \langle (\rangle\} \ .$$

> **Computing** $FIRST$
> Determining the $FIRST$ set looks very simple. While computing $FIRST(S_{i_1}\ S_{i_2}\ S_{i_3})$ it seems as if S_{i_2} and S_{i_3} can be ignored and $FIRST(S_{i_1})$ is all that matters. This may not be true. (Why?)

- Clearly, if we are given production rules

$$\begin{array}{lcl} S_{i_0} & \rightarrow & \sigma_1 \\ S_{i_0} & \rightarrow & \sigma_2 \end{array}$$

with $\langle t \rangle \in FIRST(\sigma_1) \cap FIRST(\sigma_2)$, a **predictive parser will not know what to do if the lookahead is** $\langle t \rangle$.

  - Production rules for *coll_elems*:

$$\begin{array}{lcl} \mathit{coll\_elems} & \rightarrow & \mathit{const} \\ \mathit{coll\_elems} & \rightarrow & \mathit{coll\_elems}\ \langle , \rangle\ \mathit{const} \\ \\ \mathit{const} & \rightarrow & \langle \texttt{NUM} \rangle \end{array}$$

$FIRST(\mathit{const}) \cap FIRST(\mathit{coll\_elems}\ \langle , \rangle\ \mathit{const}) = \{\langle \texttt{NUM} \rangle\}$.

- Observe that the core of our problem of common elements in $FIRST$ sets is the **left-recursion** in the production rule for $coll\_elems$.
  - A production rule is said to be **left-recursive** if it takes the following general form:

$$\begin{array}{lcl} S & \rightarrow & S\ \sigma_1 \\ S & \rightarrow & \sigma_2 \end{array}$$

    where σ_2 does not start with S.

> ✎ **Eliminating left-recursion**
> Fortunately, we can get rid of left-recursion in rules all together by a simple **rule transformation**:
>
> $$\begin{array}{lcl} S & \rightarrow & S\ \sigma_1 \\ S & \rightarrow & \sigma_2 \end{array} \quad \dashrightarrow$$

  - The modified rules pose no problem for a predictive parser.

- A similar problem arises when **two or more production rules** for the same nonterminal S **start with the same symbols**:

$$\begin{array}{lcl} \mathit{sfw} & \rightarrow & \underbrace{\texttt{SELECT}\ \mathit{proj\_list}\ \texttt{FROM}\ \mathit{from\_list}}_{=} \\ \mathit{sfw} & \rightarrow & \overbrace{\texttt{SELECT}\ \mathit{proj\_list}\ \texttt{FROM}\ \mathit{from\_list}}^{=}\ \texttt{WHERE}\ \mathit{condition} \end{array}$$

  - Only after the parser has seen the last token of the expansion of *from_list* it can decide which rule to choose ($\mathit{lookahead} = \langle\texttt{WHERE}\rangle?$).
    This does not go together with predictive one symbol lookahead parsing.

> ✎ **Left-factorization**
> Once more we can transform the production rules and make predictive parsing possible again:
>
> $$\begin{array}{lcl} S & \rightarrow & \sigma_1\ \sigma_2 \\ S & \rightarrow & \sigma_1\ \sigma_3 \end{array} \quad \dashrightarrow$$

1.2.5 `yacc` – A Parser Generator

- Once we have come up with a grammar for a language, it is simple enouh to **automatically derive a parser** from it. This is exactly what `yacc` (Yet Another Compiler Compiler) provides.

- `yacc`-generated parsers are **not** predictive but use a parsing technique called LALR(1)[6].

- `yacc`-generated parsers
  - can deal with left-recursion in rules;
  - do not need their rules left-factorized;
  - are able to use operator precedence and associativity tables to resolve conflicts during rule selection.

[6] **1**-symbol **L**ooka**h**ead, **L**eft-to-Right Parse, **R**ightmost Derivation, see [1, 2].

- A `yacc` input file comes in the general form

```
parser declarations          (tokens, operator precedence/associativity)
%%
grammar rules
%%
C program code
```

- Parser declarations:

```
1 %token OR AND EQ ...
2 %token BOOL ID SELECT FROM WHERE ...
3
4 %left     OR
5 %left     AND
6 %nonassoc EQ NEQ LEQ GEQ LT GT
7 %left     '+'
8 %left     '*'
9 %nonassoc NOT
```

- Implements this operator precedence table:

level	operator(s)
1	OR
2	AND
3	= <> <= >= < >
4	+
5	*
6	NOT

- Grammar rules:

```
1 boolean          : BOOL
2                  | boolean AND boolean
3                  | boolean OR boolean
4                  | NOT boolean
5                  | comparison
6                  | '(' boolean ')'                ;
7
8 comparison       : expr rel expr                   ;
```

✎ Shift or reduce?

The parser uses the precedence declarations to resolve conflicts like this:

$$\langle \texttt{BOOL}, \mathit{true} \rangle \ \langle \texttt{AND} \rangle \ \langle \texttt{BOOL}, \mathit{false} \rangle \underset{\rightarrow}{\curlywedge} \langle \texttt{OR} \rangle \ \langle \texttt{BOOL}, \mathit{true} \rangle$$

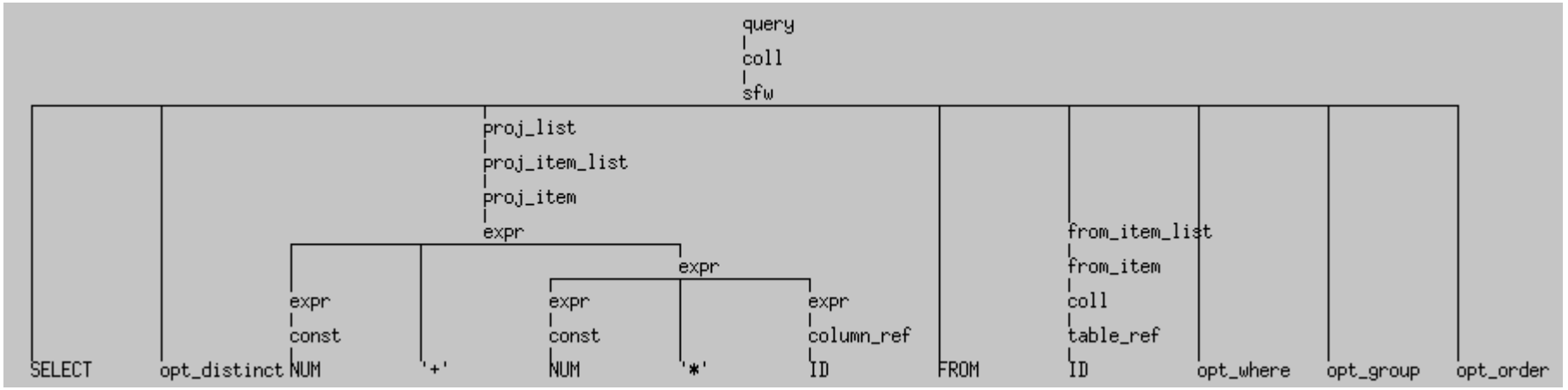
($\curlywedge$ marks how many tokens the parser has seen so far.)

✎ `yacc` **Input File**

Take a closer look at the `yacc` input file for our SQL dialect and double check the corresponding parse trees.

- Parse tree for SQL query

```
SELECT  2 + 3 * A
  FROM  R
```



1.3 Recommended Reading

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, 1986. ISBN 0-201-10088-6. The "Dragon" Book.

[2] A. W. Appel, *Modern Compiler Implementation in C: Basic Techniques*, Cambridge University Press, 1997. ISBN 0-521-58653-4.

Contents