1

Syntactic Analysis

Parsing or, more formally, *syntactic analysis* is ubiquitous document processing stage. It takes seemingly disorganized string of characters and produces distinctly structured output – a tree. Ordinary documents feature multiple levels of hierarchy: characters, words, sentences, paragraphs, chapters, volumes... From parsing perspective only the two bottom levels are significant. Document processing at the character level is called *lexical analysis*. Lexical analyzer, which is often called *scanner*, outputs a stream of words better termed as *tokens*. On the level above lexical analysis the stream of tokens is digested by a *parser* which organizes them into a *parse tree*. These two document processing phases differ greatly in their complexity, yet here our concern is usage, which in both cases reduces to easy java API call.

Scanner

Most programming languages share basically the same lexical structure. The conventional wisdom is that inventing specialized exotic lexical form for a new language has dubious value. We will, therefore, adopt "consolidated" C, C++, Java, SQL, PL/SQL lexical structure, which consists of white spaces, digits, single- and double-quoted strings, single- and multi-line comments, and operation symbols. These token types are enumerated in *Token.java*.

Scanner is run with a single call:

```
List<LexerToken> out = parse(input);
```

For example, given an input string

```
String input = "select emp from empno";
```

the scanner produces

```
0    [0,6)   select <IDENTIFIER>
1    [7,10)  emp    <IDENTIFIER>
2    [11,15) from   <IDENTIFIER>
3    [16,21) empno  <IDENTIFIER>
```

The first column contains numbers, which are token position in the list. The semi-open segment in the second column delineates beginning and end character positions of the token – its location within the input string. The next section explains it in more detail. The token context in the third column is self-explanatory. And, finally, in the last column we find the token type.

This output has been produced with java call

```
LexerToken.print(out);
```

A reader is advised to experiment with the code, by modifying and running `LexerToken.main()`.

Semi-Open Segments

Partitioning linearly ordered elements into semi-open segments became fairly standard software practice. Java text processing utilities specify intervals of characters with beginning position included, and the end position excluded. For example,

```
"select * from emp".substring(7,8);
```

cuts out wildcard "*" character fragment from the original string. We adopt this idea with somewhat unconventional notation where the beginning of the segment is denoted with square bracket and the end is parenthesis.

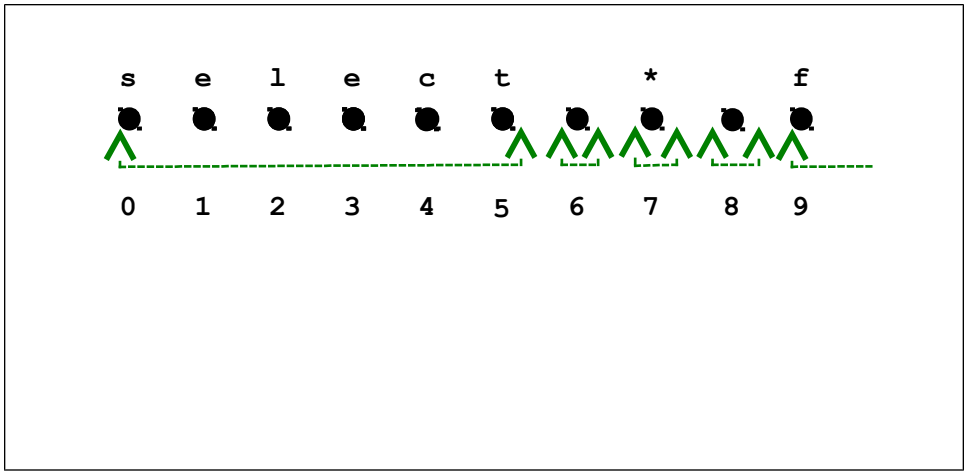

Figure 1: Semi-open segments. A sequence of characters at positions 0,1,2,3,4,5,6,7,8,9 is partitioned into segments [0,6), [6,7), [7,8), [8,9), and [9,...).

Beyond lexical analysis, we leverage semi-open segments to label nodes in the parse tree. However, parse tree segment positions are in *token* units, not character ones. For example, the parse node capturing the wildcard in the above example would be labeled

with the interval `[1,2)`. Technically, the wildcard is the third token produced by the scanner; however all whitespace tokens are ignored by parser[1].

In parser applications it is frequently required to convert from character units into token offsets, and vice versa; hence the `LexerToken.scanner2parserOffset` utility method.

Parser

Parser consumes list of tokens produced by lexer. It combines tokens according to *grammar rules*, and arranges them together with *non-terminal* grammar symbols into parse tree. The exact method how it is done constitutes rather involved computer science subject. It is the failure of the later to communicate parsing science to programmer in the trenches, that caused proliferation of crippled technologies such as XML. Parsing methods which are popular and practical admit only restricted types of grammar, while methods which allow arbitrary *context free grammars* are slow. In the first case language developer has to master arcane hacking tricks allowing him to navigate in the murky water of rule restrictions. They have to either massage the grammar to the form that particular parser generator likes or pollute grammar with side effects which guides parser engine through rule selection. No matter how cumbersome legacy parsing technologies are, they enjoy advantage of nearly instantaneous parse time, so that for users spoiled by that level of performance slow parse time of any “superior” parse method is just unacceptable. Here we claim to solve this performance vs. ability dilemma and offer a *parsing engine* which is both reasonably fast, and generic in terms of allowed grammars.

You can use parser without being required to master parsing technology. The steps are quite simple. First you instantiate parser object and feed grammar into it. For SQL&PL/SQL there is a cooked singleton already; you just get a handle to its instance as:

```
SqlEarley earley = SqlEarley.getInstance();
```

The name *Earley* refers to parsing method that we use, but you can get away with not knowing any more details. The parsing itself happens with the following calls:

```
List<LexerToken> src = ...; // from lexical analysis
Matrix matrix = new Matrix(earley);
earley.parse(src, matrix);
ParseNode root = earley.forest(src, matrix);
```

Admittedly, there is a certain amount of boilerplate code, because a reader don't have to know anything about intermediate object `matrix`. The important part is parse output shaped as a tree, which can be printed to the standard output with:

```
root.printTree();
```

For example, the input `"select emp from empno;"` produces:

```
  [0,4)   query_block  select  simple_set_expr  ...
    [0,2)   select_clause
      [0,1)   'SELECT'
```

1 Yet one more decrement is due to 0-based counting, where the “second” means object positioned at 1.

```
      [1,2)   column  expr  identifier  select_list ...
    [2,4)   from_clause
      [2,3)   'FROM'
      [3,4)   cartesian_product  identifier  query_table_expression ...
  [4,5)   ';'
```

The main purpose of the Arbori language is semantic analysis of parse trees which is described in the next chapter.

There is a convenient wrapper class which hides a great deal of boilerplate:

```
Parsed parsed = new Parsed(
            "select emp from empno;",
            SqlEarley.getInstance(),
            "sql_statements"
);
ParseNode root = parsed.getRoot();
```

The third parameter to the constructor is the expected grammar symbol. If the code fails to parse, then `parsed.getRoot()` would throw `AssertionError`. For example, the erroneous SQL in the input

```
Parsed parsed = new Parsed(
            "select from empno;",
            SqlEarley.getInstance(),
            "sql_statements"
);
ParseNode root = parsed.getRoot();
```

would output to the console:

```
***** Syntactically Invalid code fragment *****
Syntax Error
at line#0
select from empno;
       ^^^
Expected:
"aggr_name",'(','*','+','-','.',':',...,prediction,prediction_bounds,...
Exception in thread "main" java.lang.AssertionError:
  >>>> syntactically invalid code fragment <<<<
 at oracle.dbtools.parser.Parsed.getRoot(Parsed.java:67)
 at oracle.dbtools.parser.Parsed.main(Parsed.java:82)
```

Case Study: HTML/XML Parser

What about parsing anything but SQL&PL/SQL? It is just as easy; all we have to do is to provide a grammar. For readers unfamiliar with the whole parsing and languages field grammar consists of rules. Each rule can be thought of as a syntax “train diagram”, which you are undoubtedly familiar from Oracle SQL manual.

Grammar symbols can be organized into a rule in one of the two ways:

- They can be glued together sequentially, for example the rule

```
equality:  identifier '=' content
;
```

defines grammar symbol `equality` as a sequence of three symbols: `identifier`, `'='`, and `content`. Whenever a parser matches a token with an `identifier`, the next token with equality, and the following token with `value`, it matches the whole sequence of tokens with `equality`.

- A rule can branch between two or more alternatives aka choices, for example

```
opentag:     '<' identifier '>'
           |  '<' identifier equalities '>'
;
```

defines grammar symbol `opentag` as either sequence `'<' identifier '>'` or `'<' identifier equalities '>'`. In *Backus–Naur Form* sequential composition of grammar symbols takes precedence over choice.

It is remarkable that grammars don't need anything beyond sequential and alternative composition. Here is a rule which is clever combination of the two:

```
equalities:  equality
           |  equality equalities
;
```

You may notice that `equalities` is defined in terms of itself, which is still perfectly legitimate. If parser matched a sequence of tokens with `equality`, followed by sequence of tokens matching token with `equalities`, it matches the combined sequence of tokens with `equalities`. How would this process possibly be initiated? Following the first alternative: we have already seen authentic definition for `equality`, which doesn't employ recursion.

Without further ado, here is the rest of rules for our simplified XML grammar:

```
content:
            identifier
          | digits
          | '#' -- special symbols
          | content content
;

closetag:    '<' '/' identifier '>'
;

node:        opentag nodes closetag
           |  opentag content closetag
           |  opentag nodes content closetag
           |  opentag content nodes closetag
           |  opentag nodes content nodes closetag
           |  opentag closetag
;

nodes:       node
           |  nodes node
;
```

A meticulous reader may have noticed unconventional way the `content` symbol is defined. Not many parser engines allow this type of *nonlinear recursion*, but it is legitimate in our system[2].

With XML grammar at our disposal, we can proceed to parse some HTML, for example:

```
<TABLE class=TreeTableWidget border=0>
    <TR class=TableCell>
      <TD width="30%" bgColor=#bbbbbb><B>SQL_ID</B></TD>
      <TD ><B>Task 1</B></TD>
      <TD ><B>Task 2</B></TD>
    </TR>
    <TR>
      <TD><A href="http:...">684</A></TD>
      <TD>30</TD>
      <TD>42</TD>
    </TR>
    <TR>
      <TD>685</TD>
      <TD>23</TD>
      <TD>32</TD>
    </TR>
</TABLE>
```

Now everything is ready for the main program. We have written all XML grammar rules already, and let's assume they are in the text file *xml.grammar*. Our parser engine understands grammar rules presented as `Set<RuleTuple>`, where `RuleTuple` is an individual rule represented as java object. How do we get from *xml.grammar* to `Set<RuleTuple>`? We'll parse it! With the method that we have learned at the end of previous section we have:

```
Parsed xmlGrammar = new Parsed(
      Service.readFile("xml.grammar"),
       Grammar.bnfParser(),
      "grammar"
);
//xmlGrammar.getRoot().printTree();
```

Static method `Grammar.bnfParser()` provides the required parser, but where does that parser gets its set of rules? Well, we have to bootstrap somewhere, so those rules are created explicitly in java code – luckily we already described how simple *BNF grammar* is, consisting of only a handful of rules.

Now that we have a parse tree, how it's time to digest and convert it into `Set<RuleTuple>`. It is accomplished with the help of the other `Grammar` static method:

```
Set<RuleTuple> rules = new TreeSet<RuleTuple>();
Grammar.grammar(xmlGrammar.getRoot(), xmlGrammar.getSrc(), rules);
```

With XML set of rules ready we can instantiate XML parser :

```
String input = Service.readFile("test.html");
```

2 When creating rules from scratch a cautious language designer is advised to use left recursion, which greatly benefits performance of Earley method.

```
Parsed target = new Parsed(
      input,
      new Earley(rules) {
           //[3]
      },
      "nodes"
);
target.getRoot().printTree();
```

The final call outputs parse tree like this:

```
[0,149)   node  nodes
  [0,9)   opentag
    [0,1)   '<'
    [1,2)   identifier
    [2,8)   equalities
      [2,5)   equality
        [2,3)   identifier
        [3,4)   '='
        [4,5)   content  identifier
      [5,8)   equalities  equality
        [5,6)   identifier
        [6,7)   '='
        [7,8)   content  digits
    [8,9)   '>'
  [9,145)   nodes
    [9,114)   nodes
      [9,73)   node  nodes
        [9,15)   opentag
          [9,10)   '<'
          [10,11)   identifier
          [11,14)   equalities  equality
            [11,12)   identifier
            [12,13)   '='
            [13,14)   content  identifier
          [14,15)   '>'
        [15,69)   nodes
          [15,53)   nodes
            [15,37)   node  nodes
              [15,25)   opentag
                [15,16)   '<'
                [16,17)   identifier
                [17,24)   equalities
                  [17,20)   equality
                    [17,18)   identifier
                    [18,19)   '='
                    [19,20)   content  identifier
...
```

In the next chapter we'll learn how to process parse trees.

3 Here we have to influence the parser to treat double quoted strings as identifiers. This is accomplished via overriding a method within an anonymous class. See: `Parsed.main()`

2

Semantic Analysis

Parse tree is syntactic structure, and one needs to query it in order to extract useful information. This chapter describes a specialized tool for that purpose: *Arbori*[4]. From bird's eye view Arbori is a query language. Arbori query takes a parse tree as an input and outputs a relation, in the same spirit as SQL query takes tables or views as an input and outputs a view. Therefore, reader familiar with SQL would have no problem learning it. In SQL terms, each Arbori query accesses single input object – the parse tree, which allows to greatly simplify query syntax: there is no need for the FROM clause. Also the attributes restriction is implicit from the predicates or is achieved by other means, so that there is no SELECT clause. The entire query is reduced to a single WHERE clause.

Beginner Arbori Program

To give the reader a taste of the language, this section provides concise but practical example. The goal is to identify all PL/SQL assignment operators together with variable names and initialization expressions. The starting point is always looking into parse tree and figuring out the relationships between the nodes. Then, these relationships are cast formally as predicates in Arbori program.

Let's consider the following PL/SQL example:

```
BEGIN
   i:=1+2;
   x:='abc';
END;
```

which parses into

```
[0,15)   block_stmt  labeled_block_stmt
  [0,1)   'BEGIN'
  [1,13)   seq_of_stmts
    [1,8)   assignment_stmt  labeled_nonblock_stmt  sim_stmt  stmt  ...
      [1,2)   expr_id  identifier  name  name_wo_function_call
```

4 Brian Jeffries is credited for the language name.

```
      [2,3)   ':'
      [3,4)   '='
      [4,7)   and_expr  arith_expr  boolean_primary  pls_expr  rel  sim_expr
        [4,5)   arith_expr  digits  factor  numeric_literal  pri  term
        [5,6)   '+'  binary_add_op
        [6,7)   digits  factor  numeric_literal  pri  term
      [7,8)   ';'
    [8,13)   assignment_stmt  labeled_nonblock_stmt  sim_stmt  stmt ...
      [8,9)   expr_id  identifier  name  name_wo_function_call
      [9,10)   ':'
      [10,11)   '='
      [11,12)   and_expr  arith_expr  boolean_primary  factor  pls_expr  ...
      [12,13)   ';'
  [13,14)   'END'
  [14,15)   ';'
```

The two nodes of our primary interest are `[1,8)` and `[8,13)`. Both nodes are loaded with multiple grammar symbols, but it is easy to guess that we are after `assignment_stmt`. Proving, that it is indeed the right symbol requires little work. For example, to dismiss the `stmt` grammar symbol one have to look up the grammar rule (or even chain of dependent rules), and deduce that `stmt` is more general than `assignment_stmt`. Now, we can write the first line of Arbori query:

```
initializations: [assignment) assignment_stmt &
[var) name &
var^ = assignment &
var+1 = col &
[col) ':' &
col+1+1 = expr
;
```

The query name is placed before colon; hence, our query is named as **`initializations`**. Query name is followed by query definition, which in our case is conjunctive query, that is multiple conditions joined together with the **`&`** symbol. We have figured out the first predicate that the node has to be **`assignment_stmt`**. As you see from Arbori code, the syntax is close to the parse tree output. There are the same interval opening **`[`** and interval closing **`)`** brackets, but interval boundaries are no longer known. Here query language designer might have chosen to introduce variables for interval boundaries, but we have preferred to refer to the whole node. The code fragment

```
[assignment) assignment_stmt
```

is an (unary) predicate which asserts that node referred by the `assignment` variable is labeled with `assignment_stmt` grammar symbol.

Let's explain the rest of the query. The next three predicates

```
initializations: [assignment) assignment_stmt &
[var) name &
var^ = assignment &
var+1 = col &
[col) ':' &
[col+1+1) expr
;
```

introduce new node variable which we have named `var`. Its required payload – grammar symbol `name` – has been inferred after the example parse tree nodes `[1,2)` and `[8,9)`.

Here the choice of grammar symbol is little less obvious, but it doesn't matter, because we have **over-specified** this variable node with two more conditions. They assert that the node parent is the `assignment` node that we have specified before, and the next sibling node in the parse tree is named `col`. The definition of the `col` variable is perhaps the most obvious of all, because its prototype nodes in the parse tree `[2,3)` and `[9,10)` both are labeled with a single grammatical constant `':'`. The last predicate in the query asserts that the next sibling after the next sibling of `col` is some node named `expr`. It is clear that it corresponds to the parse tree nodes `[4,7)` and `[11,12)`, but by just looking onto their content it is not evident what grammar symbols we are after. The truth is that it is not important: since the `col` node is well specified (by both its payload and relative position to the `var` sibling), relative position of `expr` to the `col` specification alone is unambiguous.

Running this Arbori query against the sample PL/SQL block outputs the following relation:

```
initializations=[col        var       assignment]
                [2,3) :     [1,2) i   [1,8) "i:=1+2;"
                [9,10) :    [8,9) x   [8,13) "x:=';"
```

It have returned all the expected nodes. API for extracting this information into java environment consists of just two API calls, which we'll describe later. In the next section, we'll describe Arbor syntax in more detail.

Language Constituents

Program

Arbori program is a sequence of statements. Here is formal definition from *arbori.grammar*:

```
program:
    statement
  | program statement
;
```

A statement is a named query, that is a predicate adorned with name and trailing semicolon:

```
statement:
  identifier ':' predicate ';'
;[5]
```

Naming queries provides multiple advantages. A query following the other query, can refer to it just by name without being forced to repeat whole query expression. A query can be referred by name in java environment outside of Arbori code.

Query name is identifier, but Arbori adopts SQL convention which allows double quoted strings as identifiers.

---

5 A reader might be confused with the colon and semicolon symbols used in the two different contexts: the outer colon and semicolon are part of the standard BNF notation, while colon and semicolon literals around the `predicate` grammatical symbol are Arbori symbols.

What's in a name?

SQL identifiers syntax look odd by today's programming language standards. They are case insensitive (which by itself is not a sin); however, in what looks like a futile attempt to please everybody, many SQL implementations offer quoted identifiers.

There is more in quoted identifiers idea than just admitting names in mixed case, though. A name becomes an arbitrary string literal: the name can start with number, contain spaces, or even Kanji symbols. An identifier can be a whole sentence! Wait a minute, didn't Formal Logic and, consequently, Database Theory established already that a predicate is a sentence?

Consider the following sentence: "*Max* fed *Folly* at *2 o'clock*". This is a proposition (i.e. predicate of zero arity), which can be considered a tuple in a relation:

```
Fed=[person  pet     time]
     claire  folly    200
     max     folly    200
     max     folly    300
     max     scruffy  200
;
```

Therefore, a relation name Fed is a shorthand for "**Person** fed **pet** at **time** [o'clock]". Fully named relations may appear silly for database professional who often operates tables and views with hundreds of attributes; although, database theoretician can fire back suggesting that database design with relations of excessive arity is preposterous.

In general, it is a good programming practice to name predicates as sentences of words.

Each query is either a primitive or complex query put together with operations of conjunction, disjunction, and negation. *Atomic predicates* is a synonym which we'll often use for primitive queries.

Atomic Predicates

Atomic predicates are the most basic queries against parse tree. Arbori employs only *unary predicates* (with one node variable), or *binary predicates* (involving two nodes). Later we'll learn how to cook composite atomic predicates with logical connectives ingredients, but here we'll focus atomic predicates, first. Here is formal syntax:

```
atomic_predicate:
    '[' node ')' identifier
  | '[' node ')' string_literal
  | node '<' node
  | node '<''=' node
  | node_position '<' node_position
  | node_position '<''=' node_position
  | node '=' node
  | '?' node '=' '?' node
;
```

Each syntactic alternative is described in a dedicated subsection below.

Node grammatical content

Node content expression syntax is:

```
  '[' node ')' identifier
| '[' node ')' string_literal
```

which is reminiscent of parse tree node printout. In the following parse tree fragment

```
[0,4)   query_block  select  simple_set_expr  subquery
  [0,2)   select_clause
    [0,1)   'SELECT'
    [1,2)   '*'  select_list
```

variable `x` in the predicate `[x) select_list` matches the node `[1,2) '*' select_list` ,

Identical nodes

Often, one have node variable or expression, and would like to specify it matching the same node as the other variable or expression. The syntax for this binary relation is arguably the most intuitive out of what we have learned so far:

```
node '=' node
```

For example, knowing that `x^` is an expression denoting the parent of `x`, the predicate `x^ = y^` asserts that nodes `x` and `y` are siblings.

Ancestor-descendant

With equality relation and parent expression we can easily express that `x` is parent, or grandparent of `y`. The ancestor-descendant relation generalizes it genealogy to arbitrary number of levels. The syntax:

```
node '<''=' node
```

can be informally thought of as LHS node being older (i.e. having earlier DOB) than RHS. The strict ordering

```
node '<' node
```

disallow matching the same node.

Nodes with identical lexical content

Sometimes we want to identify nodes which may have different grammatical context, but have the same lexical content. The syntax:

```
'?' node '=' '?' node
```

For example, two nodes `x` and `y` may refer to a program language variable: `x` being declaration, and `y` being usage. This is expressed with predicate `?x = ?y`.

Nodes relative position

Remember that we have encoded each parse tree node with two numbers: beginning and the end of the segment, where the numbers refer to positions in scanned token stream. Our last primitive predicate compares those numbers at the two nodes:

```
  node_position '<' node_position
| node_position '<''=' node_position
```

where `node_position` is either beginning or the end of the segment:

```
node_position:
    '[' node
  | node ')'
;
```

For example, let `x` and `y` be the two siblings which we have learned to constrain via `x^ = y^` in the section describing the identical nodes predicate. Then, additional constraint `[x < [y` asserts that `x` is the older sibling (but not necessarily predecessor of `y`).

Operators

Given a variable referring to node in a tree, one can construct an expression referencing node's parent or siblings.

Parent

Parent node expression is the caret symbol appended to the node variable. The formal syntax:

```
node_parent:  node '^'
;
```

Consider the following parse tree fragment

```
[0,4)   query_block  select  simple_set_expr  subquery
  [0,2)   select_clause
    [0,1)   'SELECT'
    [1,2)   '*'  select_list
```

If variable `x` refers to the node `[1,2) '*' select_list` , then `x^` refers to `[0,2) select_clause`. Parent for the root node is undefined[6].

Predecessor

Node predecessor expression syntax is:

```
node_predecessor:  node '-' '1'
;
```

In the aforementioned example

```
[0,4)   query_block  select  simple_set_expr  subquery
  [0,2)   select_clause
    [0,1)   'SELECT'
    [1,2)   '*'  select_list
```

if variable `x` refers to the node `[1,2) '*' select_list` , then `x-1` refers to `[0,1) 'SELECT'`. Predecessor for the first child is undefined.

Successor

Node successor expression syntax is:

```
node_successor:  node '+' '1'
```

6 Undefined value treatment in Arbori is reminiscent of SQL NULLs

```
;
```

In our running example

```
[0,4)   query_block  select  simple_set_expr  subquery
  [0,2)   select_clause
    [0,1)   'SELECT'
    [1,2)   '*'  select_list
```

if variable `x` refers to the node `[0,1) 'SELECT'` , then `x+1` refers to `[1,2) '*' select_list`. Successor for the last child is undefined.

Boolean Connectives

Atomic predicates are combined together with standard logical connectives into [composite] predicates.

Conjunction

Conjunction is a fancy name for logical `AND` operator. The formal syntax:

```
predicate '&' predicate
```

Now we have a missing piece to finish the example from prior section, which asserted that *x is the older sibling than y*. Formally:

```
x^ = y^ & [x < [y
```

One important nuance is that the position comparison is done with numerical operator *greater-or-equal*. In our example, this allows possibility that `x` and `y` are the same node. If we want nodes to be distinct, then we write

```
x^ = y^ & x) < [y
```

Another solution – using logical negation, will be explored later.

Disjunction

Disjunction is a synonym for logical `OR` operator. The formal syntax:

```
predicate '|' predicate
```

PL/SQL grammar example provides an excellent example. Package compilation unit contains number of subprograms, and each subprogram specification contains keyword declaring either `FUNCTION` or `PROCEDURE`. Formally:

```
[f) 'PROCEDURE' | [f) 'FUNCTION'
```

Negation

Neagtion is logical `NOT` operator. The formal syntax:

```
'!' predicate
```

Equipped with negation, we can easily express condition that variables `x` and `y` refer to different tree nodes:

```
! x = y
```

Predicate variables

If you mastered SQL, you might know that often the same goal can be achieved with single long query, or be split into a sequence of smaller queries. Theoretically, long query can perform better, but in practice this is never the case. The optimizer search space grows quicker than exponential with the length of the query, therefore it is just unable to get best decisions.

In Arbori world, the performance is tied to predicate arity. Even though atomic predicates don't have more than two arguments, the list of variables may grow as long as we connect more atomic predicates. For example,

```
scope < decl & decl < id
```

is ternary predicate.

During execution of Arbori program, each variable ranges over entire parse tree[7]. Here is a simplified example illustrating nested loops evaluating a binary predicate

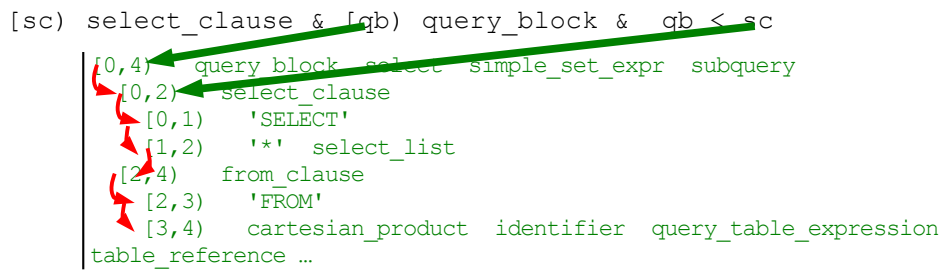

Figure 1: The query execution performance depends on predicate ranking. If Arbori starts evaluation with predicate qb < sc, then variable qb would iterate over entire tree. In the nested loop variable sc would iterate over entire tree as well. This is clearly suboptimal evaluation when, eventually, only a pair of nodes satisfying all the conditions would be found.

Arbori offers universal solution for performance problems: *divide-and-concure*. If you split your large query into statements, each having 3, max 4 node variables, the program is guaranteed to execute in milliseconds[8].

If we extracted some parse tree nodes of interest in a query, how do we reference them in later queries? This is where predicate variable syntax becomes handy:

```
pred_var :
  identifier ':' identifier[9] ';'
;
```

7 This naïve evaluation has been utilized in the early Arbori versions. Now it applies all unary predicates first – the optimization which greatly improved the performance.

8 This is no longer a restriction. Still, splitting Arbori program into easily understood statements each with limited number of variables is a good programming practice.

9 This is oversimplified syntax. Arbori allows predicate variables to refer to union of queries.

It introduces new variable corresponding to the `identifier` grammar symbol at LHS, which refers to labeled query denoted by `identifier` at RHS. Here is an example:

```
"query_blocks":
      [query_block) query_block
;

qb: "query_blocks";
```

This is a simplified fragment of larger program which queries all table/view names. It reaches the goal in small steps. The first statement queries all *query blocks* (subqueries, inner views, main select clause). For example, SQL statement

```
select * from (select * from dual);
```

contains two query blocks: `select * from dual` and `select * from (select * from dual)` itself. The `"query_blocks"` statement is followed by declaration of predicate variable `qb`, which refers to `"query_blocks"`.

This example would be incomplete without exhibiting the third statement where variable `qb` is used:

```
"table_alias" : [alias) table_reference &
qbo.query_block < alias &
...
```

Here the `alias` variable is expected to range over `table_reference` nodes inside query block extracted in the very first statement. A reader proficient in SQL or C-like programming language shouldn't have difficulty with the dot syntax: the refers to node variable `query_block` within labeled query `"query_blocks"`.

For those unconvinced that it is desirable to have dedicated predicate variables (compared to simpler world where node expression in later query is allowed to directly refer to the name of previous labeled query), here is an example:

```
"column_expr" : /* some query */ ;

ce1 : "column_expr";
ce2 : "column_expr";

join_condition :
ce1.binary_condition = ce2.binary_condition &
ce1.cexpr) < [ce2.cexpr[10]
;
```

We have declared two predicate variables referring to the same query, and used both.

Let's summarize Arbori syntax for node expression:

```
node : identifier
     | node_parent
     | node_predecessor
     | node_successor
     | referenced_node
;
```

10 This condition asserts that variables `ce1.cexpr` and `ce2.cexpr` don't refer to the same node.

Output Queries

In previous section we have advised to split complex query into a workflow of smaller lucid statements. A final statement is marked for consumption by Java program with trailing arrow `->`. Formal Arbori syntax for `statement` is:

```
statement:
    identifier ':' predicate ';'
  | identifier ':' predicate '-' '>' ';'
;
```

where the first alternative is some auxiliary query, and the second one is an output query. Let's amend previous section example

```
...

ce1 : "column_expr";
ce2 : "column_expr";

join_condition :
ce1.binary_condition = ce2.binary_condition &
ce1.cexpr) < [ce2.cexpr
->
;
```

to mark `join_condition` as output query. Arbori execution engine would follow the chain of predicate dependency and arrange predicate evaluation correspondingly. All auxiliary queries which don't affect output queries would be skipped. You have been warned: if you have forgotten to mark output queries, nothing would get evaluated!

There are two ways to deliver the results of evaluated output queries into Java.

Java Callback

Java callback is newer and less verbose method to “connect” your Arbori and Java environments. Upon firing Arbori execution you provide reference to java object

```
public class MyTest {

    static SqlProgram programInstance = null;

    public static void main( String[] args ) throws Exception {

        if( programInstance == null )

            programInstance = new SqlProgram(Service.readFile(

                                                "MyTest.prg"));

        programInstance.run("select 1 from dual", this);
```

Several other things happens in the above code snippet, so let's get them out of the way, first. The *programInstance* variable contains compiled Arbori code read from the file "MyTest.prg". Here we run Arbori code only once int the `main` method, but generally the same program is executed many times. The conditional instantiation of

programInstance is standard way to guarantee that Arbori code compilation (accomplished within `SqlProgram` constructor call) happens only once.

The last line is running Arbori program and has the following signature.

The first parameter – `"select 1 from dual"` – is SQL or PL/SQL code, which is parsed with SQL parser, and which parse tree is subject of Arbori queries. It can be read from filesystem the same way as we have read Arbori program in the preceding line.

The second parameter is the object instance. We have supplied the enclosing Java class object, but it could have been any other object. This object reference is used to perform method lookup via Java reflection, therefore the essential requirement is for this object to be equipped with methods named after Arbori program output predicates[11].

Let's complete our example

```
public class MyTest {
    static SqlProgram programInstance = null;
    public static void main( String[] args ) throws Exception {
        if( programInstance == null )
            programInstance = new SqlProgram(Service.readFile(
                                                "MyTest.prg"));
        programInstance.run("select 1 from dual",  this);
    }
    void join_condition( Parsed target, Map<String,ParseNode> tuple ) {
        System.out.println(tuple);
    }
}
```

Here we have introduced callback method `join_condition`. Again, the name is required to match the name of an output query. The method is called by Arbori execution engine per each tuple (row) produced by the output query.

The method signature is fixed. The first argument is the `Parsed` object which we focused on in chapter 1. It contains SQL input – `"select 1 from dual"` –, the lexical analysis of input as list of tokens, and parse tree. This is convenient reference whenever you want to associate a parse tree node with SQL input or list of lexemes. The second parameter is a `Map` representing each `tuple`. Inside `join_condition` method we have a single statement printing the tuple.

11 This technicality inhibits fancy quoted identifiers for output queries

Returning All Predicates

Capturing the `run` method return value is an alternative, little bit more verbose way to get Arbori query output in Java:

```
public class MyTest {
    static SqlProgram programInstance = null;
    public static void main( String[] args ) throws Exception {
        if( programInstance == null )
            programInstance = new SqlProgram(Service.readFile(
                                                "MyTest.prg"));
        Map<String,MaterializedPredicate> queries =
            programInstance.run("select 1 from dual");

        MaterializedPredicate jc = queries.get("join_condition");
        for( int i = 0; i < jc.cardinality(); i++) {
             ParseNode ce1BinaryCond = stars.getAttribute(i,
                                        "ce1.binary_condition");
             ParseNode ce2expr = stars.getAttribute(i, "ce2.cexpr");
             // ...
        }
    }
}
```

The `run` method returns a map of all query result sets, so that we can obtain any specific query result by its name. In this example, we are interested in query named `join_condition`. Somewhat awkward type name – `MaterializedPredicate` – means that this object represents a query (predicate) which has been evaluated (materialized)[12]. Then, we iterate through all the tuples explicitly in a loop. For each tuple we extract parse tree nodes that satisfy query constraints.

12 A reader familiar with JDBC may recognize `MaterializedPredicate` as analog of `ResultSet`.

Case Study: SQL Recognizer

In this section we investigate what would it take to list all the columns, tables, or other syntactic constructions. This practical application is already implemented in SQLDev *oracle.dbtools-common.jar*[13] library, so for impatient readers here is Java usage:

```
List<String> cols = SqlRecognizer.getColumns("select 1,2 from dual");
List<String> tabs = SqlRecognizer.getTables("select * from emp, dept");
```

What if you want to recognize some less ubiquitous grammar construction? The first step is obtaining a sample SQL code that faithfully represents your syntax. For example, if I were interested listing all columns in a query (and were not aware of the existing `SqlRecognizer.getColumns()` API), then SQL statement

```
SELECT POSITION,
          case when char_used = 'C' then char_length else data_length end
DATA_LENGTH,
          DATA_PRECISION, 'string literal'
FROM SYS.ALL_ARGUMENTS;
```

would be good starting point. The nodes of our interest in the parse tree are located under `[1,14)  select_list` branch between the commas:

```
   ...
         [1,14)   select_list
            [1,2)   column  expr  identifier  select_list  select_term
simple_expression
            [2,3)   ','
            [3,14)   select_term
              [3,13)   case_expression  expr
                [3,4)   'CASE'
                [4,10)   case_expression[11,16)  searched_case_expression
searched_case_expression#
                  [4,5)   'WHEN'
                  [5,8)   comparison_condition  condition
simple_comparison_condition
                    [5,6)   column  expr  identifier  simple_expression
                    [6,7)   '='  cmp_op  simple_comparison_condition[1233,1274)
                    [7,8)   expr  literal  simple_expression  string_literal
                  [8,9)   'THEN'
                  [9,10)   column  expr  identifier  simple_expression
                [10,12)   else_clause
                  [10,11)   'ELSE'
                  [11,12)   column  expr  identifier  simple_expression
                [12,13)   'END'
              [13,14)   identifier  select_term[1964,1977)
          [14,15)   ','
          [15,16)   column  expr  identifier  select_term  simple_expression
        [16,17)   ','
        [17,18)   expr  literal  select_term  simple_expression  string_literal
    [18,22)   from_clause
 ...
```

13 Formerly known as *utils-nodep.jar*

The only grammar symbol common to all four is `select_term`. Hence the Arbori program:

```
"select_term" : [column) select_term ;
```

Yes, it's just a single line of code. If you add it into *oracle/dbtools/app/recognize.prg*, then Java

```
String sql = "SELECT POSITION, ...";
Map<Integer, String> args = SqlRecognizer.fragmentAtLocation(sql,
"select_term");
```

Would give you the columns indexed by their character offsets. For example, the first column – `POSITION` – is found at offset 7.

The next step would be amending this naive Arbori program with more sophisticated relations. For example, we probably don't want columns from different query blocks bundled in a single list. The reader is encouraged to examine actual implementation *recognize.prg*, which recognizes half a dozen grammar symbols in no more than couple of pages of Arbori code.