Earley Parsing in Action

Adrian Georgescu    Vadim Tropashko

April 29, 2016

1 Introduction

Parsing is a mature technology. The field flourished with ideas in the 1960s. At that time, computer power was quite limited, this is why practical emphasis was on efficient parsing methods. Consequently, LA(1) and LALR quickly became standard; both having linear parsing time complexity.

Today, when developing a parser for new or existing language the standard programming industry approach is to take one of the readily available off-the-shelf parser engines, such as Antlr [3]. Still, many grammars (such as C) escape relatively narrow class of grammars for which there is a standard linear time parser. Various hacks are usually presented as satisfactory solutions from end-user perspective, but our general philosophy is that manual (or even automatic) transforming the grammar into LA(1), LALR, or other form admissible by a particular parser generator is inferior to leveraging more powerful (but slower) parsing method.

Parser engines evolve in this direction already. The popular bison parser has been extended to implement GLR method, which handles arbitrary context free grammar. Microsoft OSLO framework for building Domain Specific Languages (DSL) is based upon GLR parsing method as well. Likewise, Antlr parser engine evolved to accommodate more general grammar. In version 3 it leveraged LL(*) parsing that admits some grammar ambiguity. Yet LL(*) parsing is weaker than GLR, so Antlr is still not powerful to handle arbitrary context free grammar.

Both methods – GLR and LL(*) – extend an algorithm, which initially had linear parsing time. Presumably, this won’t significantly affect performance in practical cases when there is little ambiguity. In worst case, however, such an approach can result in exponential time.

There are at least two more methods that can also handle arbitrary context-free grammars: CYK and Earley. CYK [4] parsing algorithm was invented as a proof that arbitrary CFG grammar can be parsed in the time cubic by the size of the input. It is a remarkably simple idea, as core of the algorithm essentially contains only 4 nested loops. There is no automata theory, no states, no backtracking. The method provides clean prescription how to handle parse failure.

Shieber et.al. [2] describe a unified framework for both CYK and Earley. Consequently, we were able to implement Earley in the spirit of CYK method, which allowed (among other things) meaningful performance comparisons between the two. The states of parsing algorithm can be visualized as cells of two dimensional matrix listing recognized grammar rules, which is valuable for
grammar debugging. The matrix perspective also makes obvious the complexity of the algorithm, as it is proportional to the number of occupied matrix cells, which prompts organization of data structures.

After describing foundations of our approach we dive into implementation detailing the algorithm optimizations. We supplant the manuscript with implementations in C# and Java, readily available at [5].

2 Foundation

Lexical Analysis is a relatively easy parsing precursor step. During this phase the input is partitioned into a list of tokens. Lexical Analysis hinges on ubiquitous ASCII encoding partitioning character set into letters, digits, operation symbols, and white spaces. Minor variations among programming language lexical structures includes comments style and identifier definitions, so that is practical to have a unified lexer easily adjustable to any particular lexer quirks. Then, binding lexer tokens to grammar terminals is straightforward.

2.1 CYK recognizer

CYK operates on grammars in Chomsky Normal Form, such that each production has no more than two symbols concatenated on the right hand side (RHS). This limitation prompts grammar rules transformation. For example, a rule

\[ M := M \ast T \]

is rewritten into

\[ M := M X \\
X := \ast T \]

where the X is auxiliary grammar symbol.

Next, each fragment of tokens - a substring of an input - is identified by a semi-open interval demarcating the start and end positions of that substring. For example, the fragment \( 7 \ast 8 \) of the input

\[ 6 + 7 \ast 8 \]

is located between positions 2 (including) and position 5 (excluding). From now on, we’ll just identify code fragments, such as \( 7 \ast 8 \), with their location in the input, that is semi-open interval \([2,5)\).

CYK algorithm associates a set of recognized grammar symbols with each code fragment or, more rigorously, the semi-open interval which cuts this code fragment. For example, it would be able to deduce that \( 7 \ast 8 \), which was cut at the interval \([2,5)\), is a multiplicative term \( M \).

Likewise, it would fail to recognize anything for the fragment \(+ 7 \ast\). It is common wisdom to arrange recognized symbols into matrix cells which columns are enumerated with semi-open-interval start positions, and rows are interval endpoints. The table 1 exhibits the recognition matrix in our running example.

The CYK algorithm starts filling up the content of matrix cells corresponding to intervals of unit length. In our example it recognizes the term \( T \) at
Table 1: CYK matrix recognizing the multiplicative term $M$ at the interval $[2,5)$.

<table>
<thead>
<tr>
<th>Y \ X</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>M</td>
<td></td>
<td></td>
<td>T</td>
</tr>
</tbody>
</table>

Table 2: Deriving term $M$ at the interval $[2,5)$ from grammar symbol $M$ recognized at $[2,3)$ and $X$ recognized at $[3,5)$.

<table>
<thead>
<tr>
<th>Y \ X</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>T</td>
<td>M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>M</td>
<td>←</td>
<td>X</td>
<td>T</td>
</tr>
</tbody>
</table>

The crux of CYK method is splitting an interval $[X,Y)$ into 2 adjacent intervals $[X,S)$ and $[S,Y)$, and then looking for grammar rules which would combine symbols derived at the first interval together with symbols derived at the second one. In our example, suppose the terminal $M$ is recognized at semi-open interval $[2,3)$. Likewise, assume the auxiliary grammar symbol $X$ is recognized at semi-open interval $[3,5)$, then the grammar rule

$$M := M \times X$$

allows deriving symbol $M$ at semi-open interval $[2,3) \cup [3,5) = [2,5)$.

2.2 CYK optimization heuristics

The appeared CYK complexity is cubic on the length of the input, and is also proportional to the number of rules. The naïve implementation of the algorithm with Boolean arrays as described on the CYK wikipedia page is too slow for such large grammars as oracle SQL/PLSQL and realistic inputs, which are often quite long. It is unfortunate that CYK has to visit every cell, because the resulting matrix of cells is sparsely populated. If we can skip those empty cells, it can bring the method complexity closer to the actual amount of parsing work to be done.

We introduce optimization heuristic based on two observations:
Figure 1: Visualization panel for CYK derivation. Blue cells over the main diagonal are empty. Below the main diagonal we have back and green cells. Green cells are empty, although green color is used to emphasize that CYK still made some effort to derive the empty content. Black cells have some content; hovering mouse pointer over them displays it in the bottom of the panel. At the picture the current mouse location is the cell \((2,6)\) that spans tokens +, 2, *, and 3. The vertical and horizontal red lines point to two cells joining which CYK derived \texttt{boolean\_primary} from.
1. Only a subset of non-empty cells would be used for parse tree generation

2. There is no nontrivial overlaps between the intervals of those cells

This is because parse tree ancestor-dependent relation is formally defined as interval containment, so that for any two nodes on the tree their intervals are either disjoint, or contain each other.

Figure 2: Visualization of CYK performance optimization heuristic. After CYK derived the `plain_subquery` at the cell [3,13] it started to skip all the cells that intersect with it. The cells which are not examined are colored in blue.

For example, consider the following input
The inner view

select 1 from dual where 1 = 2

spans the interval \([4,12)\). It is enclosed in parenthesis which span the interval \([3,13)\).

Certainly, nothing interesting can be derived at any cell which interval intersects \([4,12)\) (or even \([3,13)\)); yet CYK would examine it. The CYK heuristic explicitly instructs it to skip an interval of cells if some "significant" symbols were derived at this interval. Again the visualization panel provides vivid illustration for the idea (Figure 2). Of course, being sure that a certain cell payload is "significant", that is going to make up to the parse tree node, is a not entirely robust assertion being made when in the middle of recognition phase. It certainly works for clauses in matched parenthesis, but with some trial-and-error we were able to extend this heuristic to a significant subset of grammar symbols.

### 2.3 Earley recognizer

Both CYK and Earley fit into unifies deduction framework [2]. Unlike CYK, Earley method puts partially recognized rules into matrix cells. Consequently, it doesn’t require grammar rules transformation. Each rule is amended with a marker inscribing recognition progress. For example, when matching interval \([2,7)\) of the following input

\[
\begin{array}{cccccccccc}
1 & + & 2 & + & 3 & + & 4 & + & 5 & + & 6 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10
\end{array}
\]

against the rule

\[
S := S + M
\]

the marker is positioned immediately before the + symbol indicating that the \(S\) portion of the rule has been recognized. On the rule listing pane of the matrix visualization panel the recognized fragment of the rule is highlighted with green background, while the remaining rule symbols are colored in light steel blue:

\[
S := \textcolor{green}{S} + M
\]

There are three recognition steps: scan, predict, and complete.
Figure 3: Scan advances recognized position within the rule by one terminal symbol (such as digits) when progressing down to the neighboring cell.

2.3.1 Scan

Each scan progresses vertically one step on matrix visualization panel, that is deriving new marked rule at cell \( (i,j+1) \) from already established rule at cell \( (i,j) \). For example, the cell \( (2,3) \) contains three rules (figure 3). The last of them is expanded to show its derivation as scan, that is advancing recognized portion to the next position within the rule, thus consuming the token digits

\[ S := \text{digits} \]
2.3.2 Predict

Figure 4: Predict advances horizontally to the cell at the main diagonal instantiating new rules to recognize.

*Predict*, moves horizontally one or more cells at once. The initial cell is at arbitrary cell \([i,j]\), while the destination is always on diagonal \([j,j]\), for example from the rule
\[ S := S + M \]

established at the cell \([0,2)\) the Earley method predicted the rule

\[ M := T \]

at the cell \([2,2)\) (figure 4).

### 2.3.3 Complete

The *complete* step joins two rules established at cells \([i,k)\) and \([k,j)\), into a new rule at cell \([i,j)\). For example, given the rule

\[ S := S + M \]

established at the cell \([0,2)\), and the rule

\[ M := T \]

fully recognized at the cell \([2,3)\), then Earley derived

\[ S := S + M \]

at the cell \([0,3)\).

### 3 The algorithm

Predict, scan, and complete are performed in a sequence for each row of cells in the matrix. Let’s assume the algorithm is currently processing row \(y\). Predict iterates through cells \([0,y), [1,y), [2,y), ... , [y-1,y)\) and adds new “predicted” rules at the cell \([y,y)\). Then, scan iterates through cells \([0,y), [1,y), [2,y), ... , [y,y)\) adding recognized rules to the cells \([0,y+1), [1,y+1), [2,y+1), ... , [y,y+1)\). Finally, complete performs nested loops. The outer loop iterates through the cells \([0,y+1), [1,y+1), [2,y+1), ... , [y,y+1)\). The inner loop splits interval \([x,y+1)\) into a join of \([x,t)\) with \([t,y+1)\), where \(x \leq t < y + 1\). The algorithm progresses filling out the cell entries below the main diagonal row by row, filling in the complete lower triangle in case of successfully recognized input.
4 Performance

The fact that we leveraged similar data structure (matrix of the cells) in both CYK and Earley methods hints that their performance is similar (cubic over the input length). It is also immediate that the cubic factor comes from the complete step of the Earley algorithm, which we focus our performance effort on.
4.0.1 Skipping cell ranges

The main heuristics hinges at the same observation that there is no nontrivial overlaps between those cells which eventually made up into parse tree nodes. The visualization panel features the same blue cell areas where algorithm didn’t even attempted to derive any content. The recognition visualization for large input is reminiscent of fractal (figures 6,7).

4.0.2 Indexing matching rules

The inner complete loop is the most critical part of the algorithm. Therefore, we tailored data structure to quick joining completed rule. A cell payload, is a list of

4.0.3 Precomputed predicts

Whenever a rule

\[ R := Q \alpha \]
Figure 7: Parsing larger code fragment (Oracle PL/SQL package body) with most of the matrix cells skipped. The vast empty areas are due to the fact that deriving a "significant" rule, for example, member function, or control block, which have unambiguously demarcated beginning and ending at the interval \([x, y)\) causes skipping all the cells corresponding overlapping intervals \([x+u, y+v)\)^2.

where \(R\) and \(Q\) are some grammar symbols, and \(\alpha\) is a (possibly empty) sequence of symbols, is predicted, the rules

\[ Q := \beta \]

has to be added onto prediction list as well. This derivation proceeds until the entire transitive closure of the first rule is obtained. However, transitive closure depends on grammar only, so it can be precomputed once only.
4.0.4 Efficient scan

Scan iterates through all partially recognized rules in a cell and promotes those rules that matches the lexed token against rule’s currently expected terminal. If the cell’s content is indexed by mapping expected terminals to the rules, then selecting a set of promoted rules is efficient.

References


