Parse Earley, Parse Often: How to Parse Anything to XML

Steven Pemberton, CWI, Amsterdam

Abstract

Invisible XML, ixml for short, is a generic technique for treating any parsable format as if it were XML, and thus allowing any parsable object to be injected into an XML pipeline. Based on the observation that XML can just be seen as the description of a parse-tree, any document can be parsed, and then serialised as XML. The parsing can also be undone, thus allowing roundtripping.

This paper discusses issues around grammar design, and in particular parsing algorithms used to recognise any document, and converting the resultant parse-tree into XML, and gives a new perspective on a classic algorithm.

Originally presented at XML London 2016, London, UK, http://xmllondon.com/2016/xmllondon-2016-proceedings.pdf#page=120 (pdf).

Introduction

What if you could see everything as XML? XML has many strengths for data exchange, strengths both inherent in the nature of XML markup and strengths that derive from the ubiquity of tools that can process XML. For authoring, however, other forms are preferred: no one writes CSS or Javascript in XML.

It does not follow, however, that there is no value in representing such information in XML. Invisible XML [ixml1, ixml2], is a generic technique for treating any (context-free) parsable document as if it were XML, enabling authors to write in a format they prefer while providing XML for processes that are more effective with XML content. There is really no reason why XML cannot be more ubiquitous than it is.

Ixml is based on the observation that XML can just be seen as the description of a parse-tree, and so any document can be parsed, and serialised as XML.

Thus can a piece of CSS such as

```
body {color: blue; font-weight: bold}
```

with ixml be read by an XML system as

```
<css>
   <rule>
      <selector>body</selector>
      <block>
         <property>
            <name>color</name>
            <value>blue</value>
         </property>
         <property>
            <name>font-weight</name>
            <value>bold</value>
         </property>
      </block>
   </rule>
</css>
```

or as

```
<css>
   <rule>
      <selector>body</selector>
      <block>
         <property name="color" value="blue"/>
         <property name="font-weight" value="bold"/>
      </block>
   </rule>
</css>
```

depending on preference. Similarly an arithmetic expression such as

```
a×(3+b)
```

can be read as

```
<expr>
    <prod>
        <letter>a</letter>
        <sum>
            <digit>3</digit>
            <letter>b</letter>
        </sum>
    </prod>
</expr>
```

and a URL such as

```
http://www.w3.org/TR/1999/xhtml.html
```

could be read as

```
<uri>
   <scheme>http</scheme>
   <authority>
      <host><sub>www</sub><sub>w3</sub><sub>org</sub></host>
   </authority>
   <path>
      <seg>TR</seg><seg>1999</seg><seg>xhtml.html</seg>
   </path>
</uri>
```

Previous presentations on ixml have centred on the basics, the design of grammars, and round-tripping back to the original form. This paper discusses a suitable parsing algorithm, and gives a new perspective on a classic algorithm.

Parsing

Parsing is a process of taking an essentially linear form, recognising the underlying structure, and transforming the input to a form that expresses that structure. The fact that the resulting structure is represented by a tree means that converting it to XML is a relatively simple matter.

There are many parsing algorithms with different properties such as speed, run-time complexity, and space usage[aho]. One of the most popular is LL1 parsing; in particular, the "1" in this name refers to the fact that you can parse based only on the knowledge of the next symbol in the input; the fact that you don't have to look ahead in the input speeds up recognition immensely.

Consider the following example of a grammar describing a simple programming language.

A grammar consists of a number of 'rules', each rule consisting, in this notation, of a name of the rule, a colon, and a definition describing the structure of the thing so named, terminated with a full-stop. That structure can consist of one or more 'alternatives', in this notation separated by semicolons. Each alternative consists of a sequence of 'nonterminals' and 'terminals' separated by commas. Nonterminals are defined by subsequent rules; terminals, enclosed in quotes, are literal characters that have to be matched in the input.

```
program: block.
block: "{", statements, "}".
statements: statement, ";", statements; empty.
statement: if statement; while statement; assignment; call; block.
if statement: "if", condition, "then", statement, else-option.
else-option: "else", statement; empty.
empty: .
while statement: "while", condition, "do", statement.
assignment: variable, "=", expression.
call: identifier, "(", parameters, ")".
parameters: expression, parameter-tail; empty.
parameter-tail: ",", expression, parameter-tail; empty.
```

This grammar is almost but not quite LL1. The problem is that the rules 'assignment' and 'call' both start with the same symbol (an identifier), and so you can't tell which rule to process just by looking at the next symbol.

However, although the *grammar* isn't LL1, the language *is*. This can be shown by combining and rewriting the rules for assignment and call:

```
statement: if statement; while statement; assignment-or-call; block.
assignment-or-call: identifier, tail.
tail: assignment-tail; call-tail.
assignment-tail: "=", expression.
call-tail: "(", parameters, ")".
```

Now the decision on which rule to use can be taken purely based on the next symbol.

One of the reasons that LL1 parsing is popular, is that it is easy to translate it directly to a program. For instance:

```
procedure program = { block; }
procedure block = { expect("{"); statements; expect("}")}
procedure statements = {
   if nextsym in statement-starters
   then {
       statement;
       expect(";");
       statements;
   }
}
procedure statement = {
   if nextsym="if" then ifstatement;
   else if nextsym="while" then whilestatement;
   else if nextsym=identifier then assignment-or-call;
   else if nextsym="{" then block;
   else error("syntax error");
}
```

etc. (This example is much simplified from what an industrial-strength parser would do, but demonstrates the principles).

However, rather than writing the parser as code, you can just as easily write what could be seen as an interpreter for a grammar. For instance:

```
procedure parserule(alts) = {
   if (some alt in alts has nextsym in starters(alt)) then parsealt(alt);
   else if (some alt in alts has (empty(alt)) then do-nothing;
   else error("syntax error");
}
procedure parsealt(alt) = {
   for term in alt do {
      if nonterm(term) then parserule(definition(term));
      else expectsym(term);
   }
}
```

One disadvantage of LL1 parsing is that no rule may be left-recursive. For instance, if the rule for 'statements' above were rewritten as

```
statements: statements, statement, ";"; empty.
```

this could not be parsed using LL1. It is easy to see why if you convert this to code, since the procedure would be:

```
procedure statements = {
   if nextsym in statement-starts {
      statements;
      statement;
      expect (";");
   }
}
```

in other words, it would go into an infinite recursive loop for 'statements'. In the case of a statement list, there is no problem with expressing it as a right-recursive rule. However, there are cases where it matters. For example, with subtraction:

```
subtraction: number; subtraction, "-", number.
```

If we rewrite this as

```
subtraction: number; number, "-", subtraction.
```

then an expression such as

```
3-2-1
```

would mean in the first case

```
((3-2)-1)
```

and in the second

```
(3-(2-1))
```

which has a different meaning.

To overcome this problem, grammars that are to be parsed with LL1 methods must have a notation to express repetition. For instance:

```
statements: (statement, ";")*.
subtraction: number, ("-", number)*.
```

which can be translated to procedures like this:

```
procedure subtraction = {
   number;
   while (nextsym="-") do {
      skipsym;
      number;
   }
}
```

Another disadvantage of LL1 and related techniques is that there has to be an initial lexical analysis phase, where 'symbols' are first recognised and classified. If not, then the level of terminal symbols that are available to the parser are the base characters, such as letters and digits, etc., meaning that for instance if the next character is an "i" you can't tell if that starts an identifier, or the word "if".

Finally, a problem with these techniques is the need to understand the LL1 conditions, express a grammar in such a way that they are satisfied, and the need to have a checker that determines if the grammar indeed satisfies the conditions before using it. [ll1][grammar improvement].

General Parsing

To summarise the advantages of LL1 parsing: it is fast, its run-time complexity is low, proportional only to the length of the input, and it is easy to express as a program.

However the disadvantages are that it can only handle a certain class of restricted languages, it requires the author of a grammar to understand the restrictions and rewrite grammars so that they match, and it requires a lexical pre-processing stage.

To overcome these disadvantages we have to consider more general parsing techniques.

One classic example is that of Earley[earley], which can parse any grammar, for any language. (There is actually one restriction, that the language is context-free, but since we are only using grammars that express context-free languages, the issue doesn't apply here).

Although the Earley algorithm is well-known, it is apparently less-often used. One reason this may be so is because the run-time complexity of Earley is rather poor in the worst case, namely $O(n^3)$. However, what potential users who are put off by this number do not seem to realise is that the complexity is a function of the language being parsed, and not the method itself. For LL1 grammars, Earley is also O(n), just like pure LL1 parsers.

So what does Earley do?

Pseudo-parallelism

Modern operating systems run programs by having many in memory simultaneously (242 on the computer this text is being written on), and allocating brief periods of time (a fraction of a second) to each in turn. A program once allocated a slot is allowed to run until it reaches the end of its execution, its time allocation expires, or the program asks to do an operation that can't be immediately satisfied (such as accessing the disk). Programs that are ready to run are added to one or more queues of jobs, and at the end of the next time slot a program is selected from the queues, based on priority, and then allowed to run for the next slot. Because the time slots are so short by human measure, this approach gives the impression of the programs running simultaneously in parallel.

Earley operates similarly. Just as the example earlier, it is an interpreter for grammars, with the exception that when a rule is selected to be run, all of its alternatives are queued to run 'in parallel'. When an alternative is given a slot, it is allowed to do exactly one task, one of:

- note that it has nothing more to do and restart its parent (that is, terminating successfully)
- start a new nonterminal process (and wait until that completes successfully)
- match a single terminal (and requeue itself)
- note that it cannot match the next symbol, and effectively terminate unsuccessfully.

The queue contains each task with the position in the input that it is at. The queue is ordered on input position, so that earlier input positions get priority. In this way only a small part of the input stream needs to be present in memory.

There is one other essential feature: when a rule starts up, its name and position is recorded in a trace before being queued. If the same rule is later started at the same position, it is not queued, since it is either already being processed, or has already been processed: we already know or will know the result of running that task at that position. This has two advantages: one is pure optimisation, since the same identical process will never be run twice; but more importantly, this overcomes the problem with infinite recursion that we saw with LL1 parsers.

To take an example, continuing to use the earlier grammar for programs, suppose the rule `statement` is being processed at the point in the input where we have the text

```
a=0;
```

Processing `statement` mean that its alternatives get queued: namely `if statement`, `while statement`, `assignment`, `call`, and `block`.

With the exception of `assignment` and `call`, all of these fail immediately because the first symbol in the input fails to match the initial item in the alternative.

`Assignment` and `call` both have as first item '`identifier`'. `Identifier` gets queued (once) and succeeds with the input 'a', so both alternatives get requeued. Since the next symbol is '=', `call` fails, and `assignment` gets requeued (and eventually succeeds).

The structure of tasks

Each task that gets queued has the following structure:

- The name of the rule that this alternative is a part of (e.g. `statement`);
- The position in the input that it started;
- The position that it is currently at;
- The list of items in the alternative that have so far been successfully parsed, with their start position;
- The list of items still to be processed.

When a task is requeued, its important parts are its current position, and the list of items still to be processed. If the list of items still to be processed is empty, then the task has completed, successfully.

Earley

So now with these preliminaries behind us, let us look at the Earley parser (here expressed in ABC [abc]):

```
HOW TO PARSE input WITH grammar:
   INITIALISE
   START grammar FOR start.symbol grammar AT start.pos
   WHILE more.tasks:
      TAKE task
      SELECT:
         finished task:
            CONTINUE PARENTS task
         ELSE:
            PUT next.symbol task, position task IN sym, pos
            SELECT:
               grammar nonterminal sym:
                  START grammar FOR sym AT pos
               sym starts (input, pos): \Terminal, matches
                  RECORD TERMINAL input FOR task
                  CONTINUE task AT (pos incremented (input, sym))
               ELSE:
                  PASS \Terminal, doesn't match
```

So let us analyse this code line-by-line:

```
START grammar FOR start.symbol grammar AT start.pos
```

All grammars have a top-level start symbol, such as `program` for the example programming language grammar. This line adds all the alternatives of the rule for the start symbol to the queue.

```
WHILE more.tasks:
```

While the queue of tasks is not empty, we loop through the following steps.

```
TAKE task
```

Take the first task from the queue.

```
SELECT:
```

There are two possibilities: the rule has nothing more to do, and so terminates successfully, or there are still symbols to process.

```
finished task:
    CONTINUE PARENTS task
```

The task has nothing more to do, so all the parents of this task (the rules that initiated it) are requeued.

```
ELSE:
```

Otherwise this task still has to be processed.

```
PUT next.symbol task, position task IN sym, pos
```

We take the next symbol (terminal or nonterminal) that still has to be processed, and the current position in the input we are at.

```
SELECT:
```

The next symbol is either a nonterminal or a terminal.

```
grammar nonterminal sym:
    START grammar FOR sym AT pos
```

The symbol is a nonterminal, so we queue all its alternatives to be processed at the current position.

Otherwise it is a terminal:

```
sym starts (input, pos): \Terminal, matches
    RECORD TERMINAL input FOR task
    CONTINUE task AT (pos incremented (input, sym))
```

If the symbol matched the input at the current position, then we record the match in the trace, and requeue the current task after incrementing its position past the matched characters.

```
ELSE:
    PASS
```

The terminal doesn't match the next symbol in the input, and so this task doesn't get requeued, essentially terminating unsuccessfully.

The Trace

The result of a parse is the trace, which is the list, for all positions in the input where a task was (re-)started, of the tasks that were (re-)started there. This is the list that is used to prevent duplicate tasks being started at the same position, but also effectively records the results of the parse.

So for instance, here is an example trace for the very first position in the input (before dealing with any input characters), for the example grammar:

```
program[1.1:1.1]: | block
block[1.1:1.1]: | "{" statements "}"
```

Two rules have been started, one for program, and consequently one for block.

The positions have been recorded as `line-number.character-number`, and here represent the position where we started from, and the position up to which we have processed for this rule, in this case, both of them `1.1`.

After the colon are two lists, separated in this output by a vertical bar: the items in the respective rule that have already been processed (none yet in this case), and the items still to be processed.

In parsing a very simple program, namely `{a=0;}`, after parsing the first opening brace, this will be in the trace:

```
block[1.1:1.2]: "{"[:1.2] | statements "}"
```

signifying that we have parsed up to position 1.2, and that we have parsed the open brace, which ends at 1.2.

Later still, after we have parsed the semicolon we will find in the trace

```
block[1.1:1.6]: "{"[:1.2] statements[:1.6] | "}"
```

which signifies we have matched the opening brace up to position 1.2, something that matches 'statements' up to 1.6, and there only remains a closing brace to match.

And finally at the end, we will find

```
block[1.1:2.1]: "{"[:1.2] statements[:1.6] "}"[:2.1] |
```

which since the 'to do' list is empty signifies a successful parse from position 1.1 to 2.1.

Since it is only completed (sub-)parses that we are interested in, here is the complete trace of all successful sub-parses for the program `{a=0;}`:

```
1.1
1.2
   empty[1.2:1.2]: |
   statements[1.2:1.2]: empty[:1.2] |
1.3
   identifier[1.2:1.3]: "a"[:1.3] |
   variable[1.2:1.3]: identifier[:1.3] |
1.4
1.5
   assignment[1.2:1.5]: variable[:1.3] "="[:1.4] expression[:1.5] |
   statement[1.2:1.5]: assignment[:1.5] |
   expression[1.4:1.5]: number[:1.5] |
   number[1.4:1.5]: "0"[:1.5] |
1.6
   statements[1.2:1.6]: statement[:1.5] ";"[:1.6] statements[:1.6] |
   empty[1.6:1.6]: |
   statements[1.6:1.6]: empty[:1.6] |
2.1
   block[1.1:2.1]: "{"[:1.2] statements[:1.6] "}"[:2.1] |
   program[1.1:2.1]: block[:2.1] |
```

Serialising the Parse Tree

So the task of serialising the trace, is one of looking at the list in the trace for the last position in the input for a successful parse for the top level symbol of the grammar, and working from there downwards:

```
SERIALISE start.symbol grammar FROM <start position> TO <end position>
```

where the procedure SERIALISE looks like this:

```
HOW TO SERIALISE name FROM start TO end:
   IF SOME task IN trace[end] HAS (symbol task = name AND finished task AND start.position task = start):
      WRITE "<", name, ">"
      CHILDREN
      WRITE "</", name, ">"
CHILDREN:
   PUT start IN newstart
   FOR (sym, pos) IN done task:
      SELECT:
         terminal sym: WRITE sym
         ELSE:
            SERIALISE sym FROM newstart TO pos
      PUT pos IN newstart
```

For our example program, this will produce:

```
<program><block>{<statements><statement><assignment><variable><identifier>a</identifier></variable>
=<expression><number>0</number></expression></assignment></statement>;
<statements><empty></empty></statements></statements>}</block></program>
```

or to make it more readable:

```
<program>
   <block>{
      <statements>
         <statement>
            <assignment>
               <variable>
                  <identifier>a</identifier>
               </variable>=
               <expression>
                  <number>0</number>
               </expression>
            </assignment>
         </statement>;
         <statements>
            <empty></empty>
         </statements>
      </statements>}
   </block>
</program>
```

The simple task of adapting this to the exact needs of ixml as described in earlier papers is left as an exercise for the reader.

Conclusion

The power of XML has been the simple and consistent representation of data, allowing widespread interoperability.

What ixml shows is that with a simple front-end, XML can be made even more powerful by making all parsable documents accessible to the XML pipeline.

References

[abc] Leo Geurts et al. "The ABC Programmer's Handbook" Prentice-Hall (ISBN 0-13-000027-2), 1990.

[aho] Aho, AV, and Ullman, JD, "The Theory of Parsing, Translation, and Compiling", Prentice-Hall, 1972, ISBN 0139145567.

[earley] "Earley Parser", https://en.wikipedia.org/wiki/Earley_parser

[grammar improvement] JM Foster, "A Syntax Improving Program", Computer Journal, Volume 11, Issue 1, pp. 31-34, 1967

[ixml1] Steven Pemberton. “Invisible XML.” Presented at Balisage: The Markup Conference 2013, Montréal, Canada, August 6 - 9, 2013. In Proceedings of Balisage: The Markup Conference 2013. Balisage Series on Markup Technologies, vol. 10 (2013). doi:10.4242/BalisageVol10.Pemberton01.

[ixml2] Steven Pemberton. "Data just wants to be (format) neutral", Presented at XML Prague, 2016, Prague, Czech Republic, http://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf, pp 109-120.

[ll1] Steven Pemberton, "Grammar Tools", http://www.cwi.nl/~steven/abc/examples/grammar.html, 1991