Jeffrey Kegler's blog about Marpa, his new parsing algorithm, and other topics of interest

Jeffrey's personal website

Follow @jeffreykegler

Marpa resources

The Marpa website

The Ocean of Awareness blog: home page, chronological index, and annotated index.

Sat, 15 Nov 2014

Parsing: Top-down versus bottom-up

Comparisons between top-down and bottom-up parsing are often either too high-level or too low-level. Overly high-level treatments reduce the two approaches to buzzwords, and the comparison to a recitation of received wisdom. Overly low-level treatments get immersed in the minutiae of implementation, and the resulting comparison is as revealing as placing two abstractly related code listings side by side. In this post I hope to find the middle level; to shed light on why advocates of bottom-up and top-down parsing approaches take the positions they do; and to speculate about the way forward.

Top-down parsing

The basic idea of top-down parsing is as brutally simple as anything in programming: Starting at the top, we add pieces. We do this by looking at the next token and deciding then and there where it fits into the parse tree. Once we've looked at every token, we have our parse tree.

In its purest form, this idea is too simple for practical parsing, so top-down parsing is almost always combined with lookahead. Lookahead of one token helps a lot. Longer lookaheads are very sparsely used. They just aren't that helpful, and since the number of possible lookaheads grows exponentially, they get very expensive very fast.

Top-down parsing has an issue with left recursion. It's straightforward to see why. Take an open-ended expression like

```
a + b + c + d + e + f + [....]
```

Here the plus signs continue off to the right, and adding any of them to the parse tree requires a dedicated node which must be above the node for the first plus sign. We cannot put that first plus sign into a top-down parse tree without having first dealt with all those plus signs that follow it. For a top-down strategy, this is a big, big problem.

Even in the simplest expression, there is no way of counting the plus signs without looking to the right, quite possibly a very long way to the right. When we are not dealing with simple expressions, this rightward-looking needs to get sophisticated. There are ways of dealing with this difficulty, but all of them share one thing in common -- they are trying to make top-down parsing into something that it is not.

Advantages of top-down parsing

Top-down parsing does not look at the right context in any systematic way, and in the 1970's it was hard to believe that top-down was as good as we can do. (It's not all that easy to believe today.) But its extreme simplicity is also top-down parsing's great strength. Because a top-down parser is extremely simple, it is very easy to figure out what it is doing. And easy to figure out means easy to customize.

Take another of the many constructs incomprehensible to a top-down parser:

```
2 * 3 * 4 + 5 * 6
```

How do top-down parsers typically handle this? Simple: as soon as they realize they are faced with an expression, they give up on top-down parsing and switch to a special-purpose algorithm.

These two properties -- easy to understand and easy to customize -- have catapulted top-down parsing to the top of the heap. Behind their different presentations, combinator parsing, PEG, and recursive descent are all top-down parsers.

Bottom-up parsing

Few theoreticians of the 1970's imagined that top-down parsing might be the end of the parsing story. Looking to the right in ad hoc ways clearly does help. It would be almost paradoxical if there was no systematic way to exploit the right context.

In 1965, Don Knuth found an algorithm to exploit right context. Knuth's LR algorithm was, like top-down parsing as I have described it, deterministic. Determinism was thought to be essential -- allowing more than one choice easily leads to a combinatorial explosion in the number of possibilities that have to be considered at once. When parsers are restricted to dealing with a single choice, it is much easier to guarantee that they will run in linear time.

Knuth's algorithm did **not** try to hang each token from a branch of a top-down parse tree as soon as it was encountered. Instead, Knuth suggested delaying that decision. Knuth's algorithm collected "subparses".

When I say "subparses" in this discussion, I mean pieces of the parse that contain all the decisions necessary to construct the part of the parse tree that is below them. But subparses do not contain any decisions about what is above them in the parse tree. Put another way, subparses know who they are, but not where they belong.

Subparses may not know where they belong, but knowing who they are is enough for them to be assembled into larger subparses. And, if we keep assembling the subparses, eventually we will have a "subparse" that is the full parse tree. And at that point we will know both who everyone is and where everyone belongs.

Knuth's algorithm stored subparses by shifting them onto a stack. The operation to do this was called a "shift". (Single tokens of the input are treated as subparses with a single node.) When there was enough context to build a larger subparse, the algorithm popped one or more subparses off the stack, assembled a larger subparse, and put the resulting subparse back on the stack. This operation was called a "reduce", based on the idea that its repeated application eventually "reduces" the parse tree to its root node.

In handling the stack, we will often be faced with choices. One kind of choice is between using what we already have on top of the stack to assemble a larger subparse; or pushing more subparses on top of the stack instead ("shift/reduce"). When we decide to reduce, we may encounter the other kind of choice -- we have to decide which rule to use ("reduce/reduce").

Like top-down parsing, bottom-up parsing is usually combined with lookahead. For the same lookahead, a bottom-up parser parses everything that a top-down parser can handle, and more.

Formally, Knuth's approach is now called shift/reduce parsing. I want to demonstrate why theoreticians, and for a long time almost everybody else as well, was so taken with this method. I'll describe how it works on some examples, including two very important ones that stump top-down parsers: arithmetic expressions and left-recursion. My purpose here is bring to light the basic concepts, and not to guide an implementor. There are excellent implementation-oriented presentations in many other places. The Wikipedia article, for example, is excellent.

Bottom-up parsing solved the problem of left recursion. In the example from above,

```
a + b + c + d + e + f + [....]
```

we simply build one subparse after another, as rapidly as we can. In the terminology of shift/reduce, whenever we can reduce, we do. Eventually we will have run out of tokens, and will have reduced until there is only one element on the stack. That one remaining element is the subparse that is also, in fact,

our full parse tree.

The top-down parser had a problem with left recursion precisely because it needed to build top-down. To build top-down, it needed to know about all the plus signs to come, because these needed to be fitted into the parse tree above the current plus sign. But when building bottom-up, we don't need to know anything about the plus signs that will be above the current one in the parse tree. We can afford to wait until we encounter them.

But if working bottom-up solves the left recursion problem, doesn't it create a right recursion problem? In fact, for a bottom-up parser, right recursion is harder, but not much. That's because of the stack. For a right recursion like this:

```
a = b = c = d = e = f = [....]
```

we use a strategy opposite to the one we used for the left recursion. For left recursion, we reduced whenever we could. For right recursion, when we have a choice, we always shift. This means we will immediately shift the entire input onto the stack. Once the entire input is on the stack, we have no choice but to start reducing. Eventually we will reduce the stack to a single element. At that point, we are done. Essentially, what we are doing is exactly what we did for left recursion, except that we use the stack to reverse the order.

Arithmetic expressions like

```
2 * 3 * 4 + 5 * 6
```

require a mixed strategy. Whenever we have a shift/reduce choice, and one of the operators is on the stack, we check to see if the topmost operator is a multiply or an addition operator. If it is a multiply operator, we reduce. In all other cases, if there is a shift/reduce choice, we shift.

In the discussion above, I have pulled the strategy for making stack decisions (shift/reduce and reduce/reduce) out of thin air. Clearly, if bottom-up parsing was going to be a practical parsing algorithm, the stack decisions would have to be made algorithmically. In fact, discovering a practical way to do this was a far from trivial task. The solution in Knuth's paper was considered (and apparently intended) to be mathematically provocative, rather than practical. But by 1979, it was thought a practical way to make stack decisions had been found and yacc, a parser generator based on bottom-up parsing, was released. (Readers today may be more familiar with yacc's successor, bison.)

The fate of bottom-up parsing

With yacc, it looked as if the limitations of top-down parsing were past us. We now had a parsing algorithm that could readily and directly parse left and right recursions, as well as arithmetic

expressions. Theoreticians thought they'd found the Holy Grail.

But not all of the medieval romances had happy endings. And as I've described elsewhere, this story ended badly. Bottom-up parsing was driven by tables which made the algorithm fast for correct inputs, but unable to accurately diagnose faulty ones. The subset of grammars parsed was still not quite large enough, even for conservative language designers. And bottom-up parsing was very unfriendly to custom hacks, which made every shortcoming loom large. It is much harder to work around a problem in a bottom-up parser than than it is to deal with a similar shortcoming in a top-down parser. After decades of experience with bottom-up parsing, top-down parsing has re-emerged as the algorithm of choice.

Non-determinism

For many, the return to top-down parsing answers the question that we posed earlier: "Is there any systematic way to exploit right context when parsing?" So far, the answer seems to be a rather startling "No". Can this really be the end of the story?

It would be very strange if the best basic parsing algorithm we know is top-down. Above, I described at some length some very important grammars that can be parsed bottom-up but not top-down, at least not directly. Progress like this seems like a lot to walk away from, and especially to walk back all the way to what is essentially a brute force algorithm. This perhaps explains why lectures and textbooks persist in teaching bottom-up parsing to students who are very unlikely to use it. Because the verdict from practitioners seems to be in, and likely to hold up on appeal.

Fans of deterministic top-down parsing, and proponents of deterministic bottom-up parsing share an assumption: For a practical algorithm to be linear, it has to be deterministic. But is this actually the case?

It's not, in fact. To keep bottom-up parsing deterministic, we restricted ourselves to a stack. But what if we track all possible subpieces of parses? For efficiency, we can link them and put them into tables, making the final decisions in a second pass, once the tables are complete. (The second pass replaces the stack-driven see-sawing back and forth of the deterministic bottom-up algorithm, so it's not an inefficiency.) Jay Earley in 1968 came up with an algorithm to do this, and in 1991 Joop Leo added a memoization to Earley's algorithm which made it linear for all deterministic grammars.

The "deterministic grammars" are exactly the bottom-up parseable grammars with lookahead -- the set of grammars parsed by Knuth's algorithm. So that means the Earley/Leo algorithm parses, in linear time, everything that a deterministic bottom-up parser can parse, and therefore every grammar that a

deterministic top-down parser can parse. (In fact, the Earley/Leo algorithm is linear for a lot of ambiguous grammars as well.)

Top-down parsing had the advantage that it was easy to know where you are. The Earley/Leo algorithm has an equivalent advantage -- its tables know where it is, and it is easy to query them programmatically. In 2010, this blogger modified the Earley/Leo algorithm to have the other big advantage of top-down parsing: The Marpa algorithm rearranges the Earley/Leo parse engine so that we can stop it, perform our own logic, and restart where we left off. A quite useable parser based on the Marpa algorithm is available as open source.

Comments

Comments on this post can be made in Marpa's Google group, or on our IRC channel: #marpa at freenode.net. To learn more about Marpa, there's the official web site maintained by Ron Savage. I also have a Marpa web site.

posted at: 17:53 | direct link to this entry

§ § §

This is Ocean of Awareness's new home. This blog has been hosted at blogs.perl.org but I have succumbed to the lure of static blogging.