An abstract view of generalized LR parsing (extended abstract)

Peter Ljunglöf
Dept. of Computer Science
Göteborg University

September 14, 2003

The LR parsing algorithm [2] is one of the most efficient parsing algorithms for deterministic context-free grammars, and used in parser generators such as yacc and bison.

The algorithm works by converting the grammar to an LR automaton, which is then traversed while keeping track of the path in a stack, pushing and popping the stack when necessary. If the automaton is determinstic, the parsing can be done in time linear in the length of the input.

Grammars for use in natural language applications are however almost always ambiguous, which means that the determinstic LR algorithm does not work. And the naive extension of trying all possible parse stacks leads to exponential behaviour. It is possible though, to extend LR parsing to handle general context-free grammars efficiently, which have been known since the early 70's [3, 6, 4]. All these implementations can be run in time cubic in the length of the input, which is as good as one can practically get for general context-free parsing.

The implementations of generalized LR parsing are often complicated and difficult to understand. As an example, the pseudo-code for Tomita-style GLR parsing is 5 pages in [5].

In this paper we show that GLR parsing is really just a natural extension of deterministic LR parsing, by abstracting away the implementation of the non-determinstic parse stack. Two of the three possible ways to implement GLR parsing (dynamic programming [3] and graph-structured stacks [6]) fit nicely in this abstraction. The third possible implementation (recursive ascent [4]) relies heavily on recursion and memoization techniques, which is difficult to fit into this framework. A third, simpler but still efficient, data structure for GLR parsing is also described.

LR automata

An LR automaton (sometimes called an LR table) is a finite automaton, with an extra *reduction* relation telling when to pop the parse stack. A finite automaton can be described by two finite sets Σ and Q of symbols and states, an initial and a final state $q_i, q_f \in Q$, and a many-valued transition function $S \in \Sigma \times Q \to 2^Q$. To this we add a reduction relation (as a many-valued function) $R \in \Sigma \times Q \to 2^{\mathbf{N} \times \Sigma}$. Since the reductions depend on one symbol as well as a state, we say that the automaton has one symbol lookahead, and call it an LR(1) automaton.

There are standard ways for compiling a context-free grammar into an LR(1) automaton, see e.g. [1], so we will assume that the automaton is given beforehand.

Generalized LR parsing

The main idea is to introduce an abstract data type Φ of collections of parse stacks. The data type has five operations:

- $\mathit{pop} \in \mathbf{N} \times \Phi \to \Phi$, which pops a number of states off each stack in the collection. Stacks which are too short are discarded.
- $\mathit{push} \in Q \times \Phi \to \Phi$, which pushes a state onto each stack in the collection.
- $\mathit{top} \in \Phi \to Q$, is only applicable on collections where all stacks have the same top state. In this case it returns the (unique) top state.
- $\mathit{split} \in \Phi \to 2^\Phi$, partitions the stacks into collections fulfilling the requirement of the *top* operation.
- $(\cup) \in 2^\Phi \to \Phi$, which joins together all stacks into one collection.

Now we can define transition functions $\mathbf{s}, \mathbf{r} \in \Sigma \times \Phi \to \Phi$:

$$\begin{aligned} \mathbf{s}(s, \phi) &= \bigcup \{\ \mathit{push}(q, \phi') \mid \phi' \in \mathit{split}(\phi),\ q \in S(s, \mathit{top}(\phi'))\ \} \\ \mathbf{r}(s, \phi) &= \bigcup \{\ \mathbf{s}(s', \mathit{pop}(n, \phi')) \mid \phi' \in \mathit{split}(\phi),\ (n, s') \in R(s, \mathit{top}(\phi'))\ \} \end{aligned}$$

Note that since the reduction doesn't consume any input symbols, it is possible to do several reductions in a row on a given symbol. This means that each input symbol is first used for a number of stack reductions, followed by a shift, after which the symbol can be discarded. So the function $\mathbf{sr}^* \in \Sigma \times \Phi \to \Phi$, defined as $\mathbf{sr}^*(s) = \mathbf{s}(s) \circ \mathbf{r}(s)^*$, can be applied to an input symbol and a stack to get all possible stacks. Finally we can define the function $\mathbf{parse} \in \Sigma^* \times \Phi \to \Phi$ as:

$$\begin{aligned} \mathbf{parse}(s_1 \ldots s_n) &= \mathbf{sr}^*(s_n) \circ \ldots \circ \mathbf{sr}^*(s_1) \\ &= \lambda\phi.\ \mathbf{foldl}(\mathbf{sr}^*, \phi, s_1 \ldots s_n) \end{aligned}$$

where **foldl** is the standard ML function with the same name. A sequence of symbolsis recognized by the automaton, if **parse** applied to the collection of the initial stack q_i, contains a stack whose top state is the final state.

Possible implementations of stack collections

There are several possible implementations for Φ. The simplest is as a list of stacks, where the *split* function can be defined by returning a large number of singleton lists of stacks, and the union ($\cup$) can be list concatenation. This implementation is rather inefficient, and there are of course more efficient implementations, which shares common parts of stacks to avoid duplicate work.

- Following Tomita [6], one way is to store the collection of stacks as a directed graph, together with a set of pointers to nodes, representing the top states. The *split* function then simply returns each node in turn, and ($\cup$) takes the union of the node pointers.
- The dynamic programming version of Lang [3], is similar to the graph-structured stack, but the graph is a bit different. The states are stored as edges spanning a part of the input, and the stacks are all possible paths spanning the whole input. The implementations of the operations are similar to the Tomita version.
- A third possible data structure is to use a tree-shaped stack, instead of a graph. This can be implemented as a relation between states and tree-shaped stacks, $\Phi \simeq 2^{Q \times \Phi}$, or if the reader prefers, as a finite map from states to tree-shaped stacks, $\Phi \simeq Q \mapsto \Phi$. The advantage to a graph is that the data structure is simpler, and we don't have to rely on global updates of the stacks. This makes it useful for e.g. lazy evaluation. *split* returns each pair in the relation as a singleton set, and ($\cup$) merges the stacks recursively.

All three implementations are efficient in that they make the parsing polynomial in the length of the input. The graph implementations are in fact cubic, and the tree implementation is conjectured to be a factor linear slower due to the merging of stacks.

References

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Donald Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.

[3] Bernard Lang. Deterministic techniques for efficient non-deterministic parsers. In J Loeckx, editor, *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, pages 255–269. Springer-Verlag, 1974.

[4] René Leermakers. *The Functional Treatment of Parsing*. Kluwer Academic Publishers, Dordrecht, Netherlands, 1993.

[5] Klaas Sikkel. *Parsing Schemata*. Springer Verlag, 1997.

[6] Masaru Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Press, 1986.