# Overview of Parsing Using Push-Down Automata

Dick Grune
VU University Amsterdam

May 2008; DRAFT

## 1 Grammars

A grammar defines a set of strings through a generative process. Once the string has been generated, the details of the generation process which determine the structure of the string are lost, but in many applications the semantics of the string depends on this structure, Determining if a string belongs to the set generated by a given grammar is called "recognition"; retrieving its generation structure is called "parsing".

In this lecture we will be concerned only with context-free (CF) grammars, because they are by far the most important; and mainly with recognition, because given the recognition process it is usually obvious what information must be collected to do parsing.

## 2 Parsing

Parsing is done mainly using two techniques: imitating the left-most derivation process (the LL methods) and rolling back the right-most derivation process (the LR methods). Examples of other methods are left-corner, non-canonical, and parsing by intersection; we shall not go into these here.

A non-deterministic (ND) push-down automaton (PDA) for imitating the left-most derivation process is easily given. For each grammar rule $P \rightarrow A_{P_1}...A_{P_m}$ we have a transition

$$(P, \varepsilon) \rightarrow (A_{P_m}...A_{P_1}, \varepsilon)$$

(a "predict move"), and for each terminal $a$ we have a transition

$$(a, a) \rightarrow (\varepsilon, \varepsilon)$$

(an "accept move"). The parser is started with the prediction $\#S$ on the stack, and is done when the end-of-input token # is accepted.

The ND PDA for rolling back the right-most production is equally simple. For each grammar rule $P \rightarrow A_{P_1}...A_{P_m}$ we have a transition

$$(A_{P_1}...A_{P_m}, \varepsilon) \rightarrow (P, \varepsilon)$$

(a "reduce move"), and for each terminal $a$ we have a transition

$$(\varepsilon, a) \rightarrow (a, \varepsilon)$$

(a "shift move"). The parser is started with an empty stack, and is done when the stack contains $S\#$ and the input is exhausted.

Both types of PDA can be simulated trivially in $O(2^n)$, where $n$ is the length of the input, by duplicating the stack when needed. The LR ND PDA can be simulated in $O(n^3)$ by tabular methods, in which segments of the stack are put in a table in such a way that corresponding segments in different stacks are represented only once. The same applies to the LL ND PDA, provided the grammar is not left-recursive.

# 3 Deterministic Parsing

The LL PDA is deterministic if there is only one rule for each non-terminal; in that case the grammar is LL(0), and produces a singleton. In all other cases there will be more than one transition with condition $(P, \varepsilon)$, for some non-terminal $P$.

The LR PDA is never deterministic: in addition to any reduce moves, a shift move is always possible.

## 3.1 Deterministic LL Parsing

The ND LL PDA can often be made deterministic by enriching it with look-ahead. Suppose we use a $k$-token look-ahead. We can then restrict the prediction transitions to those that can produce the given look-ahead. So we need to know what a given stack can predict. In a first approximation that depends on the language of the top element $P$, $L(P)$, but if that language contains strings shorter than $k$ tokens, non-terminals lower (i.e. to the left) on the stack will contribute. To handle this, we insert sets of $k$-token strings between the elements on the stack. For each grammar rule $P \rightarrow A_{P_1} ... A_{P_m}$ we have a transition

$$(s_{m+1} P s_1, \alpha) \rightarrow (s_{m+1} A_{P_m} s_m ... s_2 A_{P_1} s_1, \alpha)$$

with $\alpha \in s_1$. The $s_i$ are computed from left to right, starting with $s_{m+1}$, using $s_i = (L(A_{P_1}) s_{i+1})|_1^k$, where $t|_i^j$ is the substring from $i$ to $j$ in the string $t$. If the resulting PDA is deterministic, the grammar is LL($k$) and a deterministic linear-time parser results.

The $s_{m+1}$s can be initialized to all subsets of $k$-token strings, but very many transitions will be generated that can never be used in an actual parsing. It is better to start with the transition $\{\#^k\}S$ for the start symbol $S$, and use only combinations of sets and non-terminals that already occur in the transition table being constructed. This requires a transitive closure algorithm, and yields a reasonable parser.

For $k = 1$ a simplification is possible. Each set $s_i$ except the top one is replaced by the set of tokens that can follow $A_{P_i}$ anywhere in the grammar, its

"FOLLOW set". These FOLLOW sets can be precomputed and need not be placed on the stack, so a simple parser results. It can easily be shown that this extension of the $s_i$s cannot make the PDA non-deterministic. If an added token $t$ made it non-deterministic, the same token would appear in the $s_i$ of another occurrence before $A_{P_i}$ (or it would not be in the FOLLOW set of $A_{P_i}$) which would make that transition non-deterministic, and the grammar was not LL(1) in the first place.

Due to unfortunate misnaming, this technique is called "strong LL(1)". This is how almost all LL(1) parser generators work. It is the most popular LL method, and can – with some care – be implemented by a set of recursive routines that show great similarity to the grammar, thus allowing easy semantic elaboration.

## 3.2 Deterministic LR Parsing

A deterministic LR parser must at all times know whether to shift or to reduce, and if an reduce is indicated, to which rule to reduce. As with the deterministic LL PDA we insert sets between the stack elements, sets of states this time. These states are the states of a ND FS automaton that describes the stack. It is easily constructed from the grammar. For each rule $P \to A_{P_1}...A_{P_m}$ we have the FS transition sequence

$$s_{P_1} A_{P_1} s_{P_2} ... s_{P_m} A_{P_m} r_P$$

where the $s_{P_i}$ are shift states and $r_P$ is a reduce state. This FSA needs to be completed with several other transitions, and the resulting FSA is non-deterministic. The subset algorithm is then used to eliminate this non-determinism. If the resulting DFA has no compound states that include both a shift state and a reduce state or that include two reduce states, the grammar is LR(0) and a deterministic linear-time parser can be constructed as follows. During parsing the transitions in the DFA are used to construct the sets of states between the stack elements. The top set will then contain only shift states, or exactly one reduce state. [This is an algorithmic description, not a PDA description; needs rewording.]

Failing this, we can introduce a $k$-token look-ahead. The states in the ND FSA are annotated with a $k$-token string, indicating for which look-ahead this state is valid; this usually requires multiple copies of the grammar rules to be made for different look-aheads. The computation of these $k$-token strings is similar to that for the LL($k$) parser, except that these strings represent information about the $k$ tokens that can follow the non-terminals, rather than the $k$ tokens its production(s) can begin with.

Again the subset algorithm is used to eliminate this non-determinism. If the DFA is conflict-free in the above sense, the grammar is LR($k$), and a linear-time parser can be constructed as above.

If it is not, no deterministic parser is possible by any means [but I can't prove that].

Two simplifications are possible, and desirable, since LR($k$) transition table can be very large; both are usually only applied to LR(1) parsers.

The first is to replace the precise look-ahead information attached to each occurrence of $s_i$ by the FOLLOW set of $A_{P_i}$. Unlike the similar operation on the LL(1) parser, this seriously reduces the power of the system, but the table is easy to compute and its size is reduced to that of the LR(0) parser. This is called "SLR(1)"; it was once popular when computers were slow and had very little memory.

The second combines all those sets of states that contain identical states into a single set; of course the look-aheads of the states will differ. This reduces the number of sets to that of the LR(0) parser, but their structure is now more complicated. If the resulting DFA is still conflict-free, the grammar is LALR(1), and a linear-time parser can be constructed as above.

It can be shown by an argument similar to the one above for strong LL(1), that this process applied to an LR(1) parser cannot introduce a shift-reduce conflict. It can, however, introduce a reduce-reduce conflict. It turns out that almost all LL(1) grammars used in practise can also be handled by the LALR(1) parser generation process.

LALR(1) parsers are the most popular ones, since they ensure unambiguousness, are linear-time and cover a very large segment of the grammars in use.