1979

Fast LP(k) Analysis

Giovanni Maria Sacco

Report Number:
79-300

https://docs.lib.purdue.edu/cstech/230

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
FAST LR(k) ANALYSIS

Giovanni Maria Sacco

Computer Science Department
Purdue University
West Lafayette, IN 47907

CSD-TR 300
April 1979
SUMMARY

A general parsing technique is presented which is, at worst, as fast as the LR(k) technique, from which is derived and, at best, considerably faster. The higher speed is obtained through reduction of stack traffic.

A practical construction algorithm for any LR(k) grammar (and for all the subclasses, SLR(k) and LALR(k), etc.) is provided, as well as formal proofs and discussion of side effects, the most interesting of which is that the Fast LR(k) machine adjusts itself to the complexity of the grammar it has to parse, reducing, for regular grammars, to a deterministic finite state automaton. This fact allows the practical embodiment of lexical rules into syntactic ones, thus avoiding their artificial separation.

\[1\] On leave from RAGMA, Srl., C.so Re Umberto, 77, Torino, Italy
1-INTRODUCTION

The key idea to this paper is that many states that are stacked during LR(k) parsing [4] [1], actually don't need to be.

This is evident when we parse a regular grammar with an LR(k) parser. We know that regular grammars are recognized by finite state automata, yet, using an LR(k) parsing, we use the full strength of a deterministic pushdown automaton to analyze it.

Which states need to be stacked, then?

Intuitively, each state in which a nonterminal input is allowed must be stacked. This because such a state is a state in which the recognition of a rule \( p_1 \) is discontinued, in order to parse a rule \( p_2 \), whose left part symbol appears in the right part of \( p_1 \). After the recognition of rule \( p_2 \), parsing for \( p_1 \) must be resumed at the state in which it was discontinued, and we have to know which state that is.

States in which no nonterminal input is allowed appear therefore to be uselessly stacked. In fact, since only terminal inputs are found, we'll never discontinue the recognition of rule \( p_1 \) in these states and therefore we don't need the resume address on the stack. This is however not true, without qualification, because the number of states that are to be unstacked upon the recognition of rule \( p_1 \) will not generally be unique in this scheme, and our machine would be incorrect.

The reason is that the recognition of rule \( p_1 \) may start in different states of the machine and follow therefore different control sequences. As the reader is aware, this happens because the left hand side symbol of rule \( p_1 \) appears in the right hand side of other rules, say \( p_2 \), \( p_3 \), ... Let \( A \) be the left hand side symbol of \( p_1 \) and \( p_2 \): \( B \rightarrow \gamma A \alpha \). If \( \text{FIRST}_j(A) = \text{FIRST}_l(w) \), \( 1 \leq j \leq k \), then one of the control sequences for \( p_1 \) and one of the control sequences for \( p_2 \) will have \( j \) states in common. Suppose that in one of these states, say \( s \), there is a nonterminal input caused by rule \( p_2 \). This state becomes relevant for the parsing of \( p_1 \) in only one of its control sequences. Therefore reductions for rule \( p_1 \), with different control sequences must pop out of the stack a different number of states. This machine is then ambiguous because, upon recognition of rule \( p_1 \), we don't know how many states are to be unstacked.

We will prove, however, that a class of grammars (FLR(k) grammars) exists, for which we can avoid stacking states in which no nonterminal input is allowed and still have consistent machines.
We will develop a general algorithm for solving FLR(k) conflicts, which can be applied to any LR grammar [3] (SLR(k), LALR(k) and LR(k) grammars). The resulting machine will generally be a modified FLR(k) machine, which we call a Tentative FLR(k) (or TFLR(k)) machine.

Philosophically, our approach will be opposite to the SLR(k), LALR(k) one: while those methods start from a simple machine and modify it in order to increase its power, the FLR(k) method starts from a machine and modifies it in order to reduce it to the minimum one needed to parse its grammar. It is conceivable that 'power increasing' methods could be attempted in constructing FLR(k) machines: this approach has not yet been studied, however.

From our method an interesting formal and practical result arises: for regular grammars in the form $A \rightarrow Bwxy, x,y \in V_T$, FLR(k) machines reduce to deterministic finite state automata. Thus FLR(k) parsers can self adjust to the complexity of the grammar they have to parse. This result allows us to treat regular grammars and a subclass of context-free grammars with a unique formal device and in the same time allows compiler writers to embody lexical into syntactic analysis and be sure that lexical analysis is not slowed down by the cost of an overpowerful parser.
2—CONSTRUCTION OF FLR(k) PARSING MACHINES.

We introduce here a class of machines, called FLR(k), closely related to LR(k) machines (we follow the LR(k) machine definition of [3] [8]) and derived from them by inhibiting the stacking of states in which no nonterminal input is allowed. We are following the intuitive (and, as we discussed, in general wrong) idea that the stacking of these states is irrelevant to the progress of the analysis. We shall introduce in the following sections consistency conditions for these machines and the class of grammars that produce consistent FLR(k) machines.

DEF 2.1—An LR(k) state in which no nonterminal input is allowed is called a transfer state.\footnote{A}

DEF 2.2—An LR(k) state in which nonterminal input is allowed is called a shift state.\footnote{A}

The algorithm to construct the FLR(k) parsing machine for a grammar G is as follows:

ALGOR 2.1—Apply the following steps:
1—Construct the LR(k) parsing machine for grammar G.
2—For each transfer state $s_1$:
   2.1—Every $\gamma$-transition from the state set
   \{$s_j | s_j$ is a $\gamma$-predecessor of state $s_1$\}
   to a state $s_1$ is labelled with $T_{s_1}$.\footnote{A}

The action $T$ corresponds to a transfer of control without pushing the current state onto the stack.

EXAMPLE 2.1
Define the grammar $G = (VN, VT, P, S)$ as
\begin{align*}
1— & S \rightarrow a B c E | \\
2— & b B c d | \\
3— & b B H d \\
4— & B \rightarrow B c f | \\
5— & b \\
6— & E \rightarrow e \\
7— & H \rightarrow d h
\end{align*}

We have the following FLR(0) parsing machine (shift states are represented as squares, and actions are enclosed in brackets). Remember that, to construct an LR(k) machine, you have to augment the grammar, by adding rule 0 in the form $S' \rightarrow S \text{ \footnote{A}}$, where $S' \notin V$ is the new
start symbol, and $\varepsilon \in \Sigma T$ is the terminator:

EXAMPLE 2.2
Define the grammar $G_2 = (VN, VT, P, S)$:
1. $S \rightarrow a B c E$
2. $B \rightarrow b B c d$
3. $B \rightarrow b B c H$
4. $B \rightarrow B c f$
5. $B \rightarrow b$
6. $E \rightarrow e$
7. $H \rightarrow h$

We have the following FLR(0) parsing machine:
3—CONDITIONS FOR CONSISTENT FLR(k) PARSING MACHINES

Intuitively, an FLR(k) parser is consistent iff the number of states that must be unstacked is unique for each reduction. We formally define this concept.

DEF 3.1—A state \( s_i \) is an immediate path predecessor state of state \( s_j \), on level \( m \), according to rule \( p \) (whose right part is \( n \) symbol long), which is written
\[
s_i = IPPS(s_j, m, p)
\]
iff one of the following conditions holds:

1— if \( m=n \), then \( s_j \) must be a \( p \)-reduce state (i.e. a state in which a reduce action for rule \( p \) is defined) and there must be a transition from \( s_i \) to \( s_j \), involving the reading of a symbol \( \gamma \in V \), and \( \gamma \) must be the \( n \)-th symbol in the right part of rule \( p \).

2— if \( 0<m<n \), then there must be a transition from \( s_i \) to \( s_j \), involving the reading of a symbol \( \gamma \in V \), and \( \gamma \) must be the \( m \)-th symbol in the right part of rule \( p \).

EXAMPLE 3.1— Referring to the machine for grammar \( G_2 \):
state 5 is an IPPS(8, 3, 4)
state 3 is an IPPS(5, 2, 4)

DEF 3.2— A reduce path for rule \( p \), \( RP(p) \), where rule \( p \) has an \( n \) symbol long right part, is a sequence of states
\[
\{ s_{p0}, \ldots, s_{pn} \}
\]
of length \( n+1 \), where

1. \( s_{p0} \) is a \( p \)-reduce state
2. \( s_{p(i+1)} \in \{ s_j = IPPS(s_{p(i)}, n-i, p) \} \).

EXAMPLE 3.2— Referring to the machine for grammar \( G_2 \):
\{ 8, 5, 3, 2 \} is an RP(4)

As a machine can have several distinct reduce paths for a rule \( p \), we'll arbitrarily number all the different reduce paths and we write \( RP_i(p) \) to mean the \( i \)-th reduce path for rule \( p \).

DEF 3.3— The value of an \( RP_i(p) \), \( VAL_i(p) \), is the number of shift state in \( RP_i(p) \) minus 1.

EXAMPLE 3.3— Referring to the \( G_2 \) machine, if \( RP_1(4)=\{ 8,5,3,2 \} \), then \( VAL_1(4)=1 \).

Therefore an FLR(k) machine is consistent if, for each rule \( p \), the value of \( p \) is unique. No ambiguity will then rise when we apply a reduce action. In relation to our starting point, LR(k) machines, we note that consistent FLR(k) machines are machines for which the intuitive idea, that states in which no nonterminal input is allowed, need not to be stacked, proves true. These machines show that the LR(k) method is more powerful than required by the generating grammar.

DEF 3.4— An FLR(k) machine is said to have an FLR(k) conflict for rule \( p \), iff \( i, j \): \( VAL_i(p)=VAL_j(p) \).
DEF 3.5- An FLR(k) machine for a grammar G is said to be consistent iff

- the LR(k) machine for G has no conflicts and
- the FLR(k) machine for G has no FLR(k) conflicts.
4-FLR(k) PARSING

We now introduce the FLR(k) parsing algorithm for FLR(k) consistent machines.

ALGORITHM 4.1- Apply the following steps:
Initialization:
1- current state is state $s_0$
2- push $s_0$ onto the pushdown
Running:
3- read a symbol $\gamma \in \Sigma$ from the input tape and lookahead a $k$ symbol long string $w$ on $VT$ in order to determine which action is to be taken in the current state
4- if no action is defined in the current state for $\gamma w$, then:
   - output(error)
   - halt the parsing
else
   apply the action
4- goto 3.

Actions are defined as follows:
1- $S s_j$ (shift $s_j$):
   i- move ahead the input head by 1 cell
   ii- push $s_j$ onto the pushdown
   iii- current state is $s_i$
2- $T s_i$ (transfer $s_i$):
   i- move ahead the input head by 1 cell
   ii- current state is $s_i$
3- $R p$ (reduce $p$):
   i- output($p$)
   ii- if $p=0$ then halt the parsing, else
   iii- pop $VAL(p)$ items from the stack
   iv- current state is the state at the top of the stack
   v- force the input of the left part symbol of rule $p$.

We note that the differences between FLR(k) and LR(k) parsing are very slight, namely the new action $T$ and the number of items to be unstacked by the reduce action.

EXAMPLE 4.1- Refer to $G_j$ (example 2.1).
We have
$VAL(1), VAL(2), VAL(3)=2$
$VAL(4)=1$
$VAL(5), VAL(6), VAL(7)=0$
therefore $M(G_i)$ is consistent.

Let's parse the string $abcefe$ with $M(G_i)$:
In the LR(0) machine for G₁ there are 15 shift states, while in the FLR(0) machine for the same grammar the number of shift states is 5. Assuming each state is entered with equal probability, on the average the FLR(0) machine should have 1/3 of the stack traffic of the LR(0) one.

We note that in our parsing only 3 states needed to be stacked, while the LR(0) parsing would have required 11 pushes. Moreover the stack alphabet Γ(FLR) for an FLR(k) machine is a subset of the stack alphabet Γ(LR) for an LR(k) machine. In our example Γ(FLR)={0,2,5,8,10}.

It's then obvious that an FLR(k) machine has less stack traffic and requires a smaller stack.
5-FLR(k) GRAMMARS

So far, we have examined FLR(k) construction and parsing, but we haven’t yet determined which class of grammars produces consistent FLR(k) machines.

We will prove that FLR(k) grammars, whose definition follows, achieve this. The FLR(k) condition is cumbersome and restrictive, but it will be used to develop a tentative construction algorithm for any LR(k) grammar.

DEF 5.1—An LR(k) grammar G=(VN,VT,P,S) is FLR(k) iff at least one of the following conditions holds:

1— no pair of derivations in the form
   \[ S \Rightarrow^* A \alpha A B \gamma w \]
   or
   \[ S \Rightarrow^* B \alpha A \sigma c w \]
   exists, where
   \[ |\alpha| < |\gamma|, |w_1|, |w_2| > 0, |\sigma| \leq k \]
   \[ B, \alpha \in VN, \sigma \in VT \]

2— there is in P no rule p in the form
   \[ A \Rightarrow A \alpha \chi \]
   such that \( \text{FIRST}_k(\chi) \in \text{VT} \)

3— if \( S \Rightarrow^* A w \) then
   \[ \text{FIRST}_k(\lambda w) \neq \text{FIRST}_k(\beta w) \]

4— there is at least one derivation in the form
   \[ S \Rightarrow^* A \alpha A B \gamma w \]
   such that \( \text{FIRST}_k(B w) = \text{FIRST}_k(\sigma w) \)

5— defining
   \[ \text{SD}(\delta, S) = \text{SD}(\beta, S) \]
   as the number of different \(*\)-derivations in the form
   \[ S \Rightarrow^* \mu \alpha A \theta_i E_i w_i \]
   where
   \[ E_i \in VN \]
   \[ \theta_i \text{ varies between } \epsilon \text{ and } \delta \lambda \]
   \( \text{FIRST}_k(\sigma \lambda - \theta_i) = \text{FIRST}_k(E_i w_i) \)
   then
   \[ \text{SD}(\delta, S) = \text{SD}(\beta, S) \]

FLR(k) grammars are a proper subclass of LR(k) grammars.

DEF 5.2—An FLR(k) grammar is strong iff at least one out of the first four conditions in def 5.1 holds. Otherwise it is weak.

It is obvious that if one of the conditions ranging from 1 to 4 is true, then condition 5 is true, while the opposite doesn’t hold. It should be understood that the function SD is equivalent to the value of a reduce path for rule A->A\(\alpha\)\(\lambda\), and that condition 5 simply states that this value must be unique. Conditions 1 to 4, on the other hand, require that, for each reduce path of that rule, all states with the same sequence number are to be of the same type (i.e., all shift or all transfer states). If it is so, the value of all the reduce paths will also be unique, and, moreover, if we construct all the reduce
paths for that rule in parallel, at any time the value of the incomplete reduce paths will be unique: this is obviously stronger than what needed by condition B. Therefore the following relation holds:

\[ \text{STRONG FLR}(k) \subseteq \text{WEAK FLR}(k) \subseteq \text{FLR}(k) \subseteq \text{LR}(k) \]

In analogy to Strong and Weak FLR(k) grammars, we define Strongly and Weakly consistent FLR(k) machines.

**DEF 5.3—** A consistent FLR(k) machine is said to be **strongly consistent** iff for all i, j and for all m

s_k \in \text{RP}_i(p) \text{ and } s_j \in \text{RP}_j(p) \text{ at level m are both shift states or transfer states. Otherwise it is said to be weakly consistent.}

We now prove that an FLR(k) machine is consistent iff it is generated by an FLR(k) grammar.

**THEOREM 5.1—** An FLR(k) machine for a grammar G=(VN,VT,P,S) is consistent iff G is an FLR(k) grammar.

**Proof:**

*IF*

Suppose G is FLR(k) and the FLR(k) machine is not consistent. Then there will be at least one rule p (whose left part symbol be A) that has at least two different reduce paths RP_i(p) and RP_j(p) such that

\[ \text{VAL}_i(p) = m^n = \text{VAL}_j(p) \]

Thus there will be at least two states s_i \in \text{RP}_i(p) \text{ and } s_j \in \text{RP}_j(p) \text{ in the form}

\[
\begin{align*}
S_w & \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \\
& \downarrow c_{\{\omega_1\}} \downarrow c_{\{\omega_2\}} \downarrow c_{\{\omega_3\}} \downarrow c_{\{\omega_4\}} \\
S_1 & \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \\
& \downarrow c_{\{\omega_1\}} \\
S_2 & \rightarrow S_1 \\
S_3 & \rightarrow S_4 \\
S_4 & \rightarrow S_5 \\
S_5 & \rightarrow S_6 \\
S_6 & \rightarrow S_7
\end{align*}
\]

where \( \{ s_k, s_j \} \in \text{RP}_i(p) \)
\( \{ s_k, s_j \} \in \text{RP}_j(p) \)
\( \{ s_q, s_i \} \in \text{RP}(p_1) \)
\( \{ s_p, s_i \} \in \text{RP}(p_2) \)

\[ \text{VAL}_i(p) = m^n = \text{VAL}_j(p) \]

\[ \text{VAL}_i(p) = m^n = \text{VAL}_j(p) \]

\[ \text{VAL}_i(p) = m^n = \text{VAL}_j(p) \]
\[ \{ s_j, s_i \} \in \mathcal{P}(p), \]
such that \( s_i \) is a shift state, while \( s_j \) is a transfer state.

We note that we must have:

\[
\begin{align*}
S & \Rightarrow \delta \alpha A \sigma B w_1 \\
S & \Rightarrow \delta \beta A \sigma c w_2
\end{align*}
\]

where

\[
\delta \beta (\text{if not } s_w \Rightarrow s_w, \text{by LR}(k) \text{ construction})
\]

\[
|\sigma| \leq k (\text{if not } B \Rightarrow \{w_1\} \text{ and } c \{w_2\} \text{ would be read in different states, by LR}(k) \text{ construction}).
\]

Thus condition 1 doesn’t hold.

Condition 2 is not true because there is a rule

\[
A \Rightarrow A \sigma \lambda_1
\]

and

\[
\text{FIRST}(c \lambda_1) \in \mathcal{V} \}
\]

Since \( \text{FIRST}(c w) = \text{FIRST}(B w_1) \), condition 3 is not true.

Condition 4 doesn’t hold, because we have no nonterminal input in state \( s_j \). Therefore \( G \) is not a Strong FLR(k) grammar.

Condition 5 is not true because \( SD(\delta, S) = \text{VAL}_i(p) \) and \( SD(\delta, S) = \text{VAL}_j(p) \) by definition. Therefore \( G \) is not an FLR(k) grammar.

### ONLY IF

Suppose the FLR(k) machine \( M \) for \( G \) is consistent, while \( G \) is not FLR(k).

Then for all \( p \) and for all \( i, j \)

\[
\text{VAL}_i(p) = \text{VAL}_j(p)
\]

If none of the conditions ranging from 1 to 4 is true, we have the following situation:

- There are at least two different paths \( \mathcal{P}(p_i) \) and \( \mathcal{P}(p_j) \) that intersect in a state \( s_i \).
- There are two paths \( \mathcal{P}_i(p) \) and \( \mathcal{P}_j(p) \) relative to a rule \( p \), in the form \( A \Rightarrow \alpha \) such that
  \[
  \mathcal{P}_i(p) \text{ intersects } \mathcal{P}(p_1) \text{ starting from state } s_i \text{ at level } m
  \]
  (relative to \( p \))
  \[
  \mathcal{P}_j(p) \text{ intersects } \mathcal{P}(p_3) \text{ starting from state } s_j \text{ at level } m
  \]
  (relative to \( p \))
- \( s_j \) is a shift state, while \( s_i \) is a transfer state.

If condition 5 doesn’t hold, then \( \text{VAL}_i(p) \neq \text{VAL}_j(p) \), against the hypothesis.

### LEMMA 5.1—An FLR(k) machine is strongly consistent iff it is generated by a Strong FLR(k) grammar.

### EXAMPLE 5.1

Referring to Example 2.1, we note that \( G_1 \) is a Weak FLR(0) grammar because only condition 5 of def 5.1 holds. \( M(G_1) \) is, as expected, a weakly consistent FLR(0) machine.

Let’s note that \( G_1 \) is a Strong FLR(1) grammar, because condition 3 of def 5.1 will hold.

### EXAMPLE 5.2

Referring to example 2.2, we note that \( G_2 \) is a Strong FLR(0) grammar, since condition 4 of def 5.1 is true. \( M(G_2) \) is a strongly consistent
EXAMPLE 5.3
Define grammar $G_3=(VN, VT, P, S)$ as

\[
S \rightarrow a \ B \ c \ E \\
     b \ B \ c \ d \\
B \rightarrow B \ c \ f \\
     b \\
E \rightarrow e
\]

$G_3$ is a Strong FLR(1) grammar, but it is not FLR(0) even if it is LR(0).


6-EQUIVALENCE OF LR(k) AND FLR(k) RELATED AUTOMATA

In this section we shall prove that an FLR(k) machine doesn't stack those and onl y those states which are useless for further analysis.

DEF 6.1—Two parsing machines $M_1$ and $M_2$ are related if

- they are generated by the same grammar
- they are consistent (conflict-free)

They are equivalent if

- they are related and
- if given the same input string $w$, either they both accept $w$ or they both reject it at the same symbol in $w$.

They are strongly equivalent if

- they are equivalent and
- while parsing $w$, they produce an identical sequence of control transferred states.

THEOREM 6.1—Two related machines $M$ and $M'$, where $M$ is FLR(k) and $M'$ is LR(k), generated by the FLR(k) grammar $G$, are strongly equivalent.

Proof:

Suppose condition 2 doesn't hold.

Then there will be two sequences of control transferred states, while parsing $w$:

- $s_0...s_k s_i ...$ for $M'$
- $s_0...s_k s_j ...$ for $M$

and $s_i \neq s_j$.

Two possibilities arise:

- transfer from $s_k$ to $s_i$ in $M'$ is achieved by direct transfer (i.e. there is a transition from $s_k$ to $s_i$, involving the input of a symbol).

Then, by construction transfer from $s_k$ to $s_i$ in $M$ is achieved by direct transfer, and $s_j = s_i$, against the hypothesis.

- transfer from $s_k$ to $s_i$ in $M'$ is achieved by indirect transfer (i.e. $s_k$ is a reduce state for a rule $p$ and the sequence is $s_0...s_i...s_k s_j...$ and $|s_k...s_j|$ is equal to the length of the right part of $p$ plus 1).

By construction, transfer from $s_k$ to $s_i$ in $M$ is achieved by indirect transfer, giving the sequence $s_0...s_i...s_k s_j...$

Since $s_i \neq s_j$, this means that we unstack more or less than $\text{VAL}(p)$ items from the stack. But this is, by construction, impossible.

It is straightforward to see that condition 1 must hold too.

Therefore FLR(k) parsers have the same properties of LR(k) parsers:

- they are deterministic
- they parse a string $w$ in time $\alpha n$, $n=|w|$

Let's note, however, that the proportionality constant $\alpha$ is smaller for FLR(k) parsers.

- they recognize an error before shifting the symbol which is not consistent with the analysis.
7-TENTATIVE FLR(k) MACHINES.

Since the FLR(k) condition is too restrictive we should develop feasible ways to solve FLR(k) conflicts.

**DEF 7.1**—A tentative FLR(k) (TFLR(k)) machine is a machine which
- doesn't have any LR(k) conflicts
- does have at least one FLR(k) conflict
- can be made consistent by local modification.

By local modification we mean modifying the reduce paths for each of the troublesome rules (those, we recall, that don't have a unique value for the reduce path) in order to eliminate the inconsistencies of the machine.

We'll have Strong or Weak TFLR(k) machines, whether local modifications in point 3 induce strong or weak local consistency.

There are two major ways to locally modify an inconsistent FLR(k) machine:
- state splitting
- replacement of transfer states by shift states.

The first method derives from the following theorem, which can be proved considering condition 3 of the FLR(k) condition (def 5.1):

**THEOREM 7.1**—If a grammar G is LR(k), \( k = k_1 \), then G is FLR(k), \( k = k_2 \) and \( k_2 \leq k_1 \).

Thus, if we have two states \( s_i \) and \( s_j \) where
- \( s_i \in \text{RP}_i(p) \) is a shift state
- \( s_j \in \text{RP}_j(p) \) is a transfer state
and VAL(p) is not unique, then we can split state \( s_i \) in at least two states \( s_{i_1} \) and \( s_{i_2} \), such that
- \( s_{i_1} \in \text{RP}_i(p) \) is a shift state
- \( s_{i_2} \in \text{RP}_i(p) \) is a transfer state.

Such a state splitting is achieved by locally increasing the lookahead length.

The other method doesn't increase the lookahead string, but suitably replaces transfer states by shift states until the machine is consistent. This will eventually change the definition of the FLR(k) machine.

**DEF 7.2**—In a TFLR(k) machine, a **shift state** \( s_i \) is a state all transitions to are labelled S and a **transfer state** \( s_j \) is a state all transitions to are labelled T.

All the proofs on equivalence with LR(k) parsers we already demonstrated will still hold, if we change the definition of VAL(p) accordingly.

**THEOREM 7.2**—An LR(k) machine is a conflict-free TFLR(k) machine.
Proof: Obvious, by construction.

We can now introduce an algorithm that can be applied to any LR(k) grammar. Informally, the proposed algorithm considers each of the troublesome rules and traces all its reduce paths. Then it modifies them in order to have equal values for all of them.

This is obtained by tracing in parallel through all the reduce paths and changing a transfer state \( s_i \) in a \( RP_i(p) \) to a shift state if there is a \( RP_j(p) \) in which state \( s_j \) is a shift state and \( s_j \) has the same sequence number in the reduce path as \( s_i \).

Obviously in this way we can actually cause other rules to become troublesome, and we'll have to repeatedly scan the rule set in order to determine whether the consistent machine has been constructed or more iterations are required. It should be understood that, while theorem 7.2 proves that the process will halt, it also shows that an LR(k) machine is a possible output (namely the worst case output).

**ALGOR 7.1**

- **Input:** an LR(k) grammar \( G \), with \( P \) rules and \( S \) symbols
- **Output:** a conflict-free TFLR(k) machine for \( G \), with \( K \) states and \( S \) symbols.

**Algorithm:**

BEGIN

apply algorithm 2.1 ; /* construction */

WHILE \( p \), for which VAL(p) is not unique DO

BEGIN

FOR \( i=1 \) TO \( P \) DO

IF VAL(p) is not unique THEN

BEGIN

/* solve conflicts for rule \( p \) */

FOR \( j=\text{length of right part of rule } p \) DOWNTO 1 DO

/* array element \( RP[i,j] \) holds the state number for the \( i \)-th reduce path for rule \( p \), at level \( j \) */

IF \( RP[1,j] \) is a shift state THEN

IF \( RP[i,j] \) is a transfer state THEN

\( RP[i,j] \) becomes a shift state

ELSE /* is already a shift state: no action */

ELSE

IF \( RP[i,j] \) is a shift state THEN

\( RP[1,j] \) becomes a shift state

/* now VAL(\( i \)) is unique, but the local modification may have caused some other VAL(p) to become multivalued */

END

FOR \( p=1 \) TO \( P \) DO

update VAL(p) ;

END

/* now the TFLR(k) machine is conflict-free */

END. •
The resulting machine will be a Strong TFLR(k), and Strong TFLR(k) machines are a subset of Weak TFLR(k) machines. However, the problem of constructing Weak TFLR(k) machines has not yet been solved and it appears that the amount of computation needed for construction will possibly be so large to be unpractical.

EXAMPLE 7.1
Refer to grammar $G_3$ (Example 5.3). As we pointed out $G_3$ is not FLR(0), although it is LR(0). We apply algorithm 7.1 to construct a TFLR(0) machine for $G_3$.

After step 1, we have the following FLR(0) machine for $G_3$:

![FLR(0) machine diagram]

This machine is inconsistent, because rule 3 has $\text{VAL}_1(3)=0$, $\text{RP}_1(3)=[11,9,8,7]$ $\text{VAL}_2(3)=1$, $\text{RP}_2(3)=[11,5,4,3]$

Applying the remaining steps of the algorithm, state 9 becomes a shift state and the machine is now conflict-free, because every $\text{VAL}(p)$ is unique.

If a state splitting technique were used, we would split state 5 into two states, increasing the lookahead for all transitions to state 5. Since $G_3$ is a Strong FLR(1) grammar, the required lookahead is 1.
The machine would then be locally modified as follows:

Note that, while algorithm 7.1 doesn't increase the number of states of the resulting machine, and therefore appears more appealing, it may, however, produce LR(k) machines as a result. In certain particularly unfavourable cases for algorithm 7.1, a state splitting algorithm may be applied with better results. The decision whether to apply a state splitting algorithm should take into account the cost of lookahead vs. its benefits. Obviously, the rules that produce, after algorithm 7.1, other inconsistencies in the machine, are the natural candidates for state splitting. If, however, these rules happen to be the most used ones, it might be questionable whether the simplification of the machine balances the cost of lookahead in a critical section. The reader should be aware that state splitting and algorithm 7.1 can be used at the same time, for different parts of the machine, and therefore a selection of the troublesome rules is always possible.
We have seen so far that an FLR(k) machine is an LR(k) machine in which irrelevant stack traffic is avoided. The notion of irrelevant stack traffic obviously depends on the grammar that generates the machine and varies from grammar to grammar according to its complexity. We define informally the complexity of an FLR(k) grammar as the ratio of shift states in its FLR(k) machine minus 1 over the number of states in its LR(k) machine.

In this section we determine the class of grammars that have the minimum complexity and which form their FLR(k) machines have. We will find that left regular grammars have 0-complexity and the FLR(k) machines generated by them are deterministic finite state automata.

**THEOREM 8.1**—A left regular grammar in the form

\[ A \rightarrow B \ x \ | \ y \]

where \(x, y\) are strings on \(VT\), is an FLR(k) grammar.

**Proof:** An LRG is surely LR(k) and condition 1 of def 5.1 is true.

We now define a new type of automaton, the Deterministic Pseudo Stack Automaton (DPSA), which may be seen as a degeneration of a DPDA. A DPSA is a DPDA in which, after initialization, the stack is only inspected, but neither popped or pushed: therefore only the stack top (which is assumed to contain the initial state \(s_0\)) is meaningful.

It is obvious that a DPSA is equivalent to a DFSA, in which indirect transitions performed to \(s_0\) are made direct.

We'll now prove that LRG's are parsed by FLR(k) DPSA's.

**THEOREM 8.2**—An FLR(k) machine \(M\) for a grammar \(G\) is a DPSA iff \(G\) is a LRG.

**Proof:**

**IF**

By construction, \(s_0\) is the only shift state of \(M\).

**ONLY IF**

Suppose \(G\) is not a LRG and \(M\) is a DPSA.

Then \(G\) must have at least one rule in one of the following forms:

\[ A \rightarrow B \ x \ C \ y \]

\[ A \rightarrow y \ C \ y \]

where \(y \in \mathcal{V}^*\), \(x, y\) are strings on \(VT\).

But then there will be at least one state \(s_i\) in which a nonterminal (namely \(C\)) input is allowed. Then \(s_i\) is a shift state and \(M\) is a DPSA.

If we define a left regular component of an LR(k) grammar \(G=(V_N, V_T, P, S)\) as the left regular grammar \(G_1=(V'_N, V'_T, P', A)\), where

\[ A \in V_N, V'_N, V_T, V'_T, P', P \]

and \(P'\) is in the form \(A \rightarrow E \ x \ | \ y, x, y \in V_T^*\),

then there exists a \(k\) for which the part of the FLR(k) machine for \(G\), that parses \(G_1\) is a finite automaton.

We will give an intuition of the correctness of this assertion. If \(G=G_1\) is empty, then \(G\) is regular and, as we proved, the FLR(k) machine is a finite automaton. Otherwise, if the part of the machine that...
parses $G_1$ is not a finite automaton, then there is at least a control path which is shared with the part of the machine parsing $G-G_1$. Therefore, increasing the lookahead will cause the elimination of this sharing. Note that, due to the necessary conditions for this sharing (def 5.1 condition 1), lexical rules in the description of artificial languages are not likely to present any trouble.

From a theoretical point of view, the FLR($k$) method can be used as a powerful test on left regular components of Context-free grammars.

We have now proved our claim that FLR($k$) machines self adjust to the complexity of the grammar they have to parse. We noted that a DPSA is equivalent to a DFSA: therefore we can use the same machine to practically parse both lexical and syntactic rules of the language.
OPTIMIZING TECHNIQUES FOR TFLR(k) PARSERS

We'll refer to the methods discussed in [8]. These are of two classes:

A-automaton structure modification
1-SLR(k) methods [2]
2-LALR(k) methods [6]
3-unit reduction elimination [7]
4-final state elimination [7]

B-automaton representation modification
1-code compaction
2-matrix compaction [9]
3-extended matrix compaction [9]

SLR(k) and LALR(k) are the only parsing methods in the LR family used for practical purposes [3]: for artificial languages, \( k = 1 \) generally proves sufficient. As we claimed before, our method is applicable to them without any modification. We are not going to formally prove this assertion, but rather give an intuition of its correctness.

Both methods use an increased lookahead length to solve RR and RS conflicts in a given machine, generally an LR(0) one. The machine then becomes a hybrid LR machine, partly LR\((k_i)\) and partly LR\((k)\), where \( k_i < k \). Stack operations however don't vary, the only difference between the machines being the lookahead length. Our method is concerned only with stack operations and it is not affected in any way by the lookahead length: as long as stack operations are performed consistently with the LR\((k)\) definition, algorithm 7.1 will work correctly, as well as state splitting techniques. The same informal proof is easily demonstrated for A.3 and A.4. Obviously we'll have to perform the TFLR\((k)\) construction after the application of any method in class A.

The following inclusion rules hold:

\[
\text{SFLR}(k) \subseteq \text{LAFLR}(k) \subseteq \text{FLR}(k) \\
\text{STFLR}(k) \subseteq \text{LATFLR}(k) \subseteq \text{TFLR}(k)
\]

One remark is to be done on final state elimination. Final state elimination causes the reduction to be performed in a final state to be anticipated, by means of a SR (shift-reduce) action. Therefore, states in which for every nonterminal input \( A \), we have a SR action associated to all \( a \in \text{VT} \) such that \( a = \text{FIRST}_1(A) \), can be considered transfer states. Note that, applying this considerations to the machine for \( G_1 \), we will produce a consistent FLR\((k)\) machine.

Methods in class B simply modify the resulting automaton representation (i.e. the coding of the transition function) and are therefore to be use after the TFLR\((k)\) construction.

The TFLR\((k)\) method can then be practically used for constructing optimized parsers and, if algorithm 7.1 is used, the number of states for a TFLR\((k)\) and an LR\((k)\) machine generated by the same grammar will be the same and the TFLR\((k)\) parsing speed will be, at worst, as fast
as the LR(k) one.

ACKNOWLEDGMENTS

I am indebted to Prof. C. Hoffmann and to Prof. M. O'Donnell, whose careful readings of this paper have greatly improved its clarity and accuracy.

REFERENCES


[9] SACCO, G.M. —"On the Reduced Matrix Representation of LR(k) Parsers" manuscript