Programming Languages
D. Gries Editor

Simple $LR(k)$ Grammars

Franklin L. DeRemer
University of California, Santa Cruz

A class of context-free grammars, called the "Simple $LR(k)$" or $SLR(k)$ grammars is defined. This class has been shown to include weak precedence and simple precedence grammars as proper subsets. How to construct parsers for the $SLR(k)$ grammars is also shown. These parser-construction techniques are extendible to cover all of the $LR(k)$ grammars of Knuth; they have been implemented and by direct comparison proved to be superior to precedence techniques, not only in the range of grammars covered, but also in the speed of parser construction and in the size and speed of the resulting parsers.

Key Words and Phrases: context-free grammar, $LR(k)$ grammar, precedence grammar, syntactic analysis, parsing algorithm, parser, finite-state machine, deterministic pushdown automaton

CR Categories: 4.12, 5.22, 5.23

Work reported herein was supported in part by Project MAC, an MIT research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01).

1. Introduction

Some of the results attained in the course of thesis research by the author are reported in this paper. We define a class of context-free grammars, called the "Simple $LR(k)$" or $SLR(k)$ grammars, that has been shown [2] to include the weak precedence grammar [9] and the simple precedence grammar [14]. We also show how to construct parsers for $SLR(k)$ grammars. The construction technique can be extended [2, 4] to an algorithm for constructing a parser for any $LR(k)$ grammar [11]; i.e. any grammar which generates strings each of which can be parsed during a single deterministic scan from left (L) to right (R) without looking ahead more than k symbols.

Our parser-constructing techniques have been implemented [3, 7], and they have been compared with the mixed strategy precedence (MSP) techniques of McKeeman [12]. It was found that in addition to the main advantage of working for a larger class of grammars, our $SLR(k)$ techniques are superior to the precedence techniques both in the speed of parser construction and in the size and speed of the parsers produced. Our $SLR(1)$ techniques require from one-tenth to one-twentieth the time required by MSP techniques to construct parsers for practical grammars, and the resulting $SLR(1)$ parsers require about two-thirds the space and time required for the corresponding MSP parsers.

2. Terminology

A *context-free* (CF) *grammar* is a quadruple (V_T, V_N, S, P) where V_T is a finite set of symbols called *terminals*, V_N is a finite set of symbols distinct from those in V_T called *nonterminals*, S is a distinguished member of V_N called the *starting symbol*, and P is a

finite set of pairs called *productions*. Each production is written $A \rightarrow \omega$ and has a *left part* A in V_N, and a *right part* ω in V^*, where $V = V_N \cup V_T$. V^* denotes the set of all strings composed of symbols in V, including the empty string.

Without loss of generality we assume that one production is of the form $S \rightarrow \vdash S' \dashv$, where S' is a subordinate starting symbol and S and the terminal "pad" symbols $\vdash$ and $\dashv$ appear in none of the other productions. We use Latin capitals to denote nonterminals, lowercase Latin letters and special symbols (e.g. +, *, :, etc.) to denote terminals, and lowercase Greek letters to denote strings. We use $|\beta|$ to denote the length of (number of symbols in) the string β, and $k{:}\beta$ to denote the first k symbols of β if $|\beta| \geq k$, and β otherwise.

In the sequel, we often use for examples the grammar

$$G_1 = (\{(,), i, \uparrow, +, \vdash, \dashv\}, \{S, E, T, P\}, S, P_1)$$

where P_1 consists of the following productions:

$$\begin{array}{lll} S \rightarrow \vdash E \dashv & T \rightarrow P \uparrow T & P \rightarrow i \\ E \rightarrow E + T & T \rightarrow P & P \rightarrow (E) \\ E \rightarrow T & & \end{array}$$

If $A \rightarrow \omega$ is a production, an *immediate derivation* of one string $\alpha = \rho\omega\beta$ from another $\alpha' = \rho A\beta$ is written $\alpha' \rightarrow \alpha$. The transitive completion of this relation is a *derivation* and is written $\alpha' \rightarrow^* \alpha$, which means there exist strings $\alpha_0, \alpha_1, \cdots, \alpha_n$ such that $\alpha' = \alpha_0 \rightarrow \alpha_1 \rightarrow \cdots \rightarrow \alpha_n = \alpha$ for $n \geqq 0$. We choose as our *canonical derivation* the right derivation, i.e. the derivation in which each step is of the form $\rho A\beta \rightarrow \rho\omega\beta$ where β is in V_T^*.

A *terminal string* is one consisting entirely of terminals. A *sentential form* is any string derivable from S. A *sentence* is any terminal sentential form. The *language* $L(G)$ generated by a CF grammar G is the set of sentences, i.e. $L(G) = \{\eta \in V_T^* \mid S \rightarrow^* \eta\}$. We assume that G has no *useless* productions; i.e. we assume that for each production $A \rightarrow \omega$ there exists a derivation $S \rightarrow^* \sigma A\beta \rightarrow \sigma\omega\beta \rightarrow^* \sigma\delta\beta$, where σ, δ, and β are (possibly null) terminal strings. Well-known methods exist for detecting and removing useless productions [8].

Loosely speaking, a *parse* of a string is some indication of how that string was derived. In particular, a *canonical parse* of a sentential form α is the reverse of the sequence of productions used in a canonical derivation of α. We refer to the action of determining a parse as *parsing*, and a parsing algorithm is called a *parser*.

3. Parsers for LR(0) Grammars

In this section we briefly review the results of Knuth [11] regarding $LR(0)$ grammars. We use a combination of the terminologies of Earley [6] and McKeeman [12].

To construct an $LR(0)$ parser for a CF grammar $G = (V_T, V_N, S, P)$, we compute *configuration sets*.

Each member of a configuration set is called a *configuration* and it is a production in P with a special *marker* (we use a dot ".") in its right part. There is an *initial configuration set*, namely $S_0 = \{S \rightarrow . \vdash S' \dashv\}$, and the other sets are computed as indicated below.

Intuitively, each configuration set represents a possible "state of the parse." It will be seen below that our parsing algorithm will successively enter "states," each of which corresponds to a configuration set. When the parser is in a state corresponding to a certain configuration set, the configurations in that set indicate which parts of which productions may have been used to generate the input string at the point currently of interest. Initially, the parser will be in a state corresponding to the initial configuration set; i.e. the "state of the parse" at the beginning may be summarized by saying that the string must have been generated by first using the production $S \rightarrow \vdash S' \dashv$ and that the first symbol to be read must be $\vdash$, as indicated by the marker to its left in the configuration of the initial set.

Each nonempty configuration set has one or more *successors*: other configuration sets. For instance, S_0 has a single successor, called a "$\vdash$-successor", which contains the configuration $S \rightarrow \vdash . S' \dashv$, among others. In general, a configuration set S_i has an *s-successor* for each symbol s in V that is preceded by a marker in one or more of S_i's configurations.

Successor configuration sets correspond to successor "states of the parse." If the parser is in a state corresponding to a set having a configuration with a marker before the symbol s, and if the next symbol to be read *is* an s, then the parser will read the s and enter a state corresponding to the s-successor of the original state.

This s-successor consists of a *basis set* unioned with a *closure set*. The basis set consists of all configurations in S_i having a marker before an s, but with the marker moved to follow the s; i.e. it is $\{A \rightarrow \omega s . \omega' \mid A \rightarrow \omega . s\omega' \in S_i\}$. The closure set is defined recursively to be the largest set of configurations of the form $A \rightarrow . \omega$ such that $A \rightarrow \omega$ is in P and there exists a configuration with a marker before an A in either the basis set or the closure set.

We add the configurations in the closure set to the "state of the parse" because, if a configuration has a marker before a nonterminal, we must also have all the productions defining that nonterminal in the set. This is true because we cannot read a nonterminal directly if we are parsing only terminal strings, but must instead read a whole phrase and reduce that phrase to that nonterminal. We reiterate that the "state of the parse," i.e. a configuration set, indicates which parts of which productions may have been used to generate the input string at a given point of interest to the parser.

In the special case of a configuration with a marker to the right of all symbols in the right part of the production, the corresponding successor is the empty set and is called the "$\#_{A \rightarrow \omega}$-successor" where $A \rightarrow \omega$ is the production involved.

The notion of a $\#_{A \rightarrow \omega}$-successor is related to a "state of

Fig. 1. The configuration sets and successor relations of the $LR(0)$ parser for our example grammar G_1.

State names (numbers)	Configuration sets	Successor relations
0	$\{S \rightarrow .\vdash E \dashv\}$	$\xrightarrow{\vdash}$ 1
1	$\{S \rightarrow \vdash .E \dashv$ $E \rightarrow .E + T\}$	$\xrightarrow{E}$ 2
	$E \rightarrow .T$	$\xrightarrow{T}$ 6
	$T \rightarrow .P \uparrow T$ $T \rightarrow .P\}$	$\xrightarrow{P}$ 7
	$P \rightarrow .i$	$\xrightarrow{i}$ 10
	$P \rightarrow .(E)\}$	$\xrightarrow{(}$ 11
2	$\{S \rightarrow \vdash E .\dashv$	$\xrightarrow{\dashv}$ 3
	$E \rightarrow E. + T\}$	$\xrightarrow{+}$ 4
3	$\{S \rightarrow \vdash E \dashv .\}$	$\#S \rightarrow \vdash E \dashv$ 14
4	$\{E \rightarrow E + .T$	$\xrightarrow{T}$ 5
	$T \rightarrow .P \uparrow T$ $T \rightarrow .P\}$	$\xrightarrow{P}$ 7
	$P \rightarrow .i$	$\xrightarrow{i}$ 10
	$P \rightarrow .(E)\}$	$\xrightarrow{(}$ 11
5	$\{E \rightarrow E + T.\}$	$\#E \rightarrow E+T$ 14
6	$\{E \rightarrow .T\}$	$\#E \rightarrow T$ 14
7	$\{T \rightarrow P. \uparrow T$ $T \rightarrow P.\}$	$\xrightarrow{\uparrow}$ 8 $\#T \rightarrow P$ 14
8	$\{T \rightarrow P \uparrow .T$	$\xrightarrow{T}$ 9
	$T \rightarrow .P \uparrow T$ $T \rightarrow .P\}$	$\xrightarrow{P}$ 7
	$P \rightarrow .i$	$\xrightarrow{i}$ 10
	$P \rightarrow .(E)\}$	$\xrightarrow{(}$ 11
9	$\{T \rightarrow P \uparrow T.\}$	$\#T \rightarrow P \uparrow T$ 14
10	$\{P \rightarrow i.\}$	$\#P \rightarrow i$ 14
11	$\{P \rightarrow (.E)$ $E \rightarrow .E + T\}$	$\xrightarrow{E}$ 12
	$E \rightarrow .T$	6
	$T \rightarrow .P \uparrow T$ $T \rightarrow .P\}$	$\xrightarrow{P}$ 7
	$P \rightarrow .i$	$\xrightarrow{i}$ 10
	$P \rightarrow .(E)\}$	$\xrightarrow{(}$ 11
12	$\{P \rightarrow (E.)$ $E - E. + T\}$	$\xrightarrow{)}$ 13 $\xrightarrow{+}$ 4
13	$\{P \rightarrow (E).\}$	$\#P \rightarrow (E)$ 14
14	$\{\ \}$	

the parse" that indicates that a reduction should be made of a substring ω to the nonterminal A; this is explained via the parsing algorithm given below.

An $LR(0)$ *parser* for a grammar G, then, is represented by the set of all configuration sets computed by starting with S_0 and adding to the set all successors of members already in the set.

As an example, we begin the computation of the $LR(0)$ parser for our example grammar G_1. As usual,

$S_0 = \{S \rightarrow .\ \vdash E \dashv\}$.

S_0 has only a $\vdash$-successor, call it S_1, whose basis set is

$\{S \rightarrow \vdash .\ E \dashv\}$

and whose closure set is

$\{E \rightarrow .\ E + T,$
$E \rightarrow .\ T,$
$T \rightarrow .\ P \uparrow T,$
$T \rightarrow .\ P,$
$P \rightarrow .\ i,$
$P \rightarrow .\ (E)\}$.

S_1 has an E-successor (call it S_2), a T-successor (S_6), a P-successor (S_7), an i-successor (S_{10}), and a (-successor (S_{11}). Set S_2 has the basis set

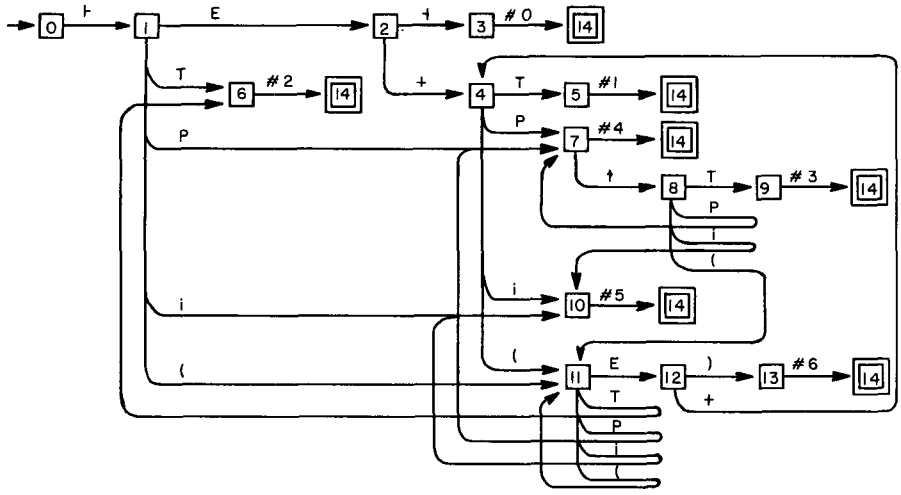
$\{S \rightarrow \vdash E .\ \dashv, E \rightarrow E .\ + T\}$

and an empty closure set. The entire $LR(0)$ parser is indicated in Figure 1.

From a set of configuration sets and their successor relations we can abstract the essential structure and get a finite state machine (FSM). For each configuration set there is a corresponding state in the FSM; the empty configuration set corresponds to the final state. The transitions of the FSM correspond to the successor relations. This FSM has been called [2] the *characteristic FSM* (or CFSM) of the grammar. The CFSM of grammar G_1 is illustrated in Figure 2.

Any CFSM state with transitions under symbols in V only is called a *read state*. Any state with one transition under one of the special $\#_{A \rightarrow \omega}$-symbols and zero or one transition under a nonterminal symbol is called a *reduce* state. States having two or more $\#_{A \rightarrow \omega}$-transitions

Fig. 2. The CFSM of our example grammar G_1 : (0) $S \rightarrow \vdash E \dashv 5$ (1) $E \rightarrow E + T$, (2) $E \rightarrow T$, (3) $T \rightarrow P \uparrow T$, (4) $T \rightarrow P$, (5) $P \rightarrow 1$, (6) $P \rightarrow (E)$. The 14 in the double box denotes a single state, the final state. The subscripts on the #-symbols reference productions.



or having one or more $\#_{A \rightarrow \omega}$-transitions and one or more transitions under terminal symbols are called *inadequate* states.

In our terminology, Knuth showed that a CF grammar G is $LR(0)$ if and only if its CFSM has no inadequate states. (Thus grammar G_1 is not $LR(0)$.) As is shown below, this means that a parser can be constructed for G which scans deterministically from left to right and which always knows whether to read another symbol or to stop and make a reduction, and in the latter case which reduction to make—all without having to investigate any symbols to the right of the points in strings at which such decisions must be made. Also, Knuth showed that the following parsing algorithm is correct for an $LR(0)$ grammar G.

LR(0) Parsing Algorithm. Maintain a stack on which to store alternately symbols in V and the names cf states entered by G's CFSM. We begin with the CFSM in state S_0 (corresponding to the initial configuration set) and with the name S_0 the only item in the stack. To parse a string η in $L(G)$:

1. Let $\alpha = \eta$.
2. Starting the CFSM in whatever state it was in prior to this step, cause the CFSM to begin reading α and to store on the stack each symbol read followed by the name of the state entered subsequently.[1]
3. When the CFSM enters a reduce state (which it must do sooner or later), set α equal to the suffix of α not yet read. Let the production associated with the reduce state be $A \rightarrow \omega$. Pop the top $2 *| \omega |$ items off the stack. If $A = S$ the parse is complete, so stop; otherwise return the CFSM to the state whose name is on top of the stack, set $\alpha = A\alpha$, and go to step (2).

Since the parsing algorithm uses a stack and since the CFSM constitutes a finite control, our parser is equivalent to a deterministic pushdown automaton (DPDA, see [8]).

It has been shown in [2] and in [11] that, if this algorithm is applied to a string not in $L(G)$, the CFSM will at some point enter a read state that has no transition under the next symbol to be read, i.e. it has been shown that the algorithm fails for strings not in $L(G)$.

The history of the above algorithm using the CFSM of Figure 2 and applied to the string $\eta = \vdash i + i \dashv$ in $L(G_1)$ is indicated in Table I. Note that the decisions whether to read or reduce at lines 4 and 9 were made somehow magically since the associated state 7 is inadequate.

Intuitive Explanation. Since each CFSM state-name corresponds to a configuration set, it is as if the parsing algorithm were storing configuration sets between symbols on the stack. The configuration sets merely represent the "state of the parse" at various points in the string.

For instance, the state of the parse at the beginning is represented by $\{S \rightarrow . \vdash E \dashv\}$, indicating that we are expecting an instance of an S which is composed of a $\vdash$ followed by an E followed by a $\dashv$. The marker "." before the $\vdash$ indicates that the next thing to be matched is a $\vdash$.

If $\vdash$ *is* the next symbol in the string, we read it and proceed to a state represented by $\{S \rightarrow \vdash . E \dashv, E \rightarrow . E + T, E \rightarrow . T, T \rightarrow . P \uparrow T, T \rightarrow . P, P \rightarrow . i, P \rightarrow . (E)\}$; otherwise the string is in error (not in $L(G)$). The latter set indicates that the next symbol must be an

[1] Actually the symbols need not be stacked; they can be deduced from the state-names. Each state is entered after reading a unique symbol; note that each marker in a given configuration set is preceded either by no symbol (members of the closure set) or by one specific symbol (members of the basis set). Thanks go to J. J. Horning for observing this uniqueness.

E, which may be in the form of an $E + T$ or a T, and the T may be in the form $P \uparrow T$ or P, and the P may be in the form i or (E). So, we may expect an E, T, P, i, or (, next.

If the next symbol is in fact an i, it is clear that the pertinent configuration was $P \rightarrow . i$. Thus we read the i and proceed to a "state of the parse" represented by $\{P \rightarrow i . \}$, indicating that we have just read the right part of the production and may replace it with the left part.

Upon doing so, we return to the state of the parse which existed before reading the right part (the i) and find out what to do when the left part (P) is the next symbol. In the case at hand, the pertinent configurations are $T \rightarrow . P \uparrow T$ and $T \rightarrow . P$, so we read the P and proceed to the state represented by $\{T \rightarrow P . \uparrow T, T \rightarrow P . \}$.

The latter set corresponds to an inadequate state and our parsing algorithm, as it stands now, fails.

4. Parsers for Simple LR(k) Grammars

When our CFSM enters an inadequate state we do not know whether to stop and make a reduction or to allow the CFSM to continue reading. The notion of a Simple $LR(k)$ grammar arises from a particular, simple solution to the indecisiveness associated with inadequate states. We first consider $SLR(1)$ grammars and then generalize to $SLR(k)$ grammars.

SLR(1) Grammars

A CF grammar G is said to be $SLR(1)$ if and only if each of the inadequate states of its CFSM has mutually disjoint *simple 1-look-ahead sets* associated with its terminal- and #-transitions.

A simple 1-look-ahead set is associated with each transition from an inadequate state. For a transition under a symbol s in V, the set is $\{s\}$. For a transition under a symbol $\#_{A \rightarrow \omega}$, where $A \rightarrow \omega$ is in P, the set is $F_T^1(A) = \{s \in V_T \mid S \rightarrow^* \rho A s \beta$ for some $\rho, \beta\}$, i.e. the set of terminal symbols which may follow the nonterminal A in some sentential form.

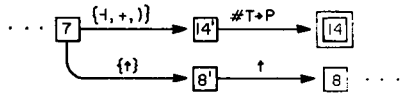
As an example we compute $F_T^1(P)$ for grammar G_1 : P appears in the right parts of two productions. The production $T \rightarrow P \uparrow T$ implies that $\uparrow$ is in $F_T^1(P)$. The production $T \rightarrow P$ implies that all the strings in $F_T^1(T)$ are also in $F_T^1(P)$. $E \rightarrow E + T$ and $E \rightarrow T$ each imply that the members of $F_T^1(E)$ are also in $F_T^1(T)$. $S \rightarrow \vdash E \dashv$ implies that $\dashv$ is in $F_T^1(E)$; $E \rightarrow E + T$ adds $+$; and $P \rightarrow (E)$ adds ")". Thus, we have determined that $F_T^1(P) = \{\uparrow, \dashv, +,)\}$, and in the process that $F_T^1(T) = F_T^1(E) = \{\dashv, +,)\}$. In the Appendix we give a method for finding simple 1-look-ahead sets directly from the CFSM.

Grammar G_1 is $SLR(1)$ since the inadequate state 7 of its CFSM has the disjoint simple 1-look-ahead sets: $\{\uparrow\}$ for the $\uparrow$-transition and $F_T^1(T) = \{\dashv, +,)\}$ for the $\#_{T \rightarrow P}$-transition.

To get a parser for an $SLR(1)$ grammar we must modify (a) the CFSM and (b) the parsing algorithm.

(a) Each inadequate state N of the CFSM is replaced by a *look-ahead state* N' such that, for each transition from N to some state M under a symbol s and with associated simple 1-look-ahead set L, there exists a transition from N' to some new state M' under the set L, and from M' there is exactly one transition, namely one under s to state M.

The appropriately modified state 7 for the CFSM of grammar G_1 is illustrated below.



(b) Our parsing algorithm must be modified in two ways. First, it must treat look-ahead states properly: if and when the CFSM enters a look-ahead state N, investigate but do not read the next symbol s and cause the CFSM to enter next the state to which goes the transition under the look-ahead set containing s. Second, the algorithm must never push on its stack the name of any of the new states (M' of above) to which go transitions from look-ahead states.

Table I. The history of the CFSM of Figure 2 applied to the string $\eta = \vdash i + i \dashv$

Stack	Input	Output	Line No.
$_0$	$\vdash i + i \dashv$		1
$_0\vdash_1$	$i + i \dashv$		2
$_0\vdash_1 i_{10}$	$+ i \dashv$		3
		$P \rightarrow i$	
$_0\vdash_1 P_7$	$+ i \dashv$		4†
		$T \rightarrow P$	
$_0\vdash_1 T_6$	$+ i \dashv$		5
		$E \rightarrow T$	
$_0\vdash_1 E_2$	$+ i \dashv$		6
$_0\vdash_1 E_2 +_4$	$i \dashv$		7
$_0\vdash_1 E_2 +_4 i_{10}$	$\dashv$		8
		$P \rightarrow i$	
$_0\vdash_1 E_2 +_4 P_7$	$\dashv$		9†
		$T \rightarrow P$	
$_0\vdash_1 E_2 +_4 T_5$	$\dashv$		10
		$E \rightarrow E + T$	
$_0\vdash_1 E_2$	$\dashv$		11
$_0\vdash_1 E_2 \dashv_3$			12
		$S \rightarrow \vdash E \dashv$	
$_0$			fini

† Note that there is some magic in lines 4 and 9.

The magic in lines 4 and 9 of Table I can now be explained. For example, we detail below the actions of the modified parsing algorithm using the modified CFSM to accomplish the task of line 4 in Table I.

CFSM State	*Stack*	*Input*	*Output*	*Line No.*
7	$_0\vdash_1 P_7$	$+\ i\ \dashv$		4
14′	$_0\vdash_1 P_7$	$+\ i\ \dashv$	$T \rightarrow P$	4′
6	$_0\vdash_1 T_6$	$+\ i\ \dashv$		5

Generalization to SLR(k)

Conceptually, the generalization from $SLR(1)$ to $SRL(k)$ is simple. Instead of using $F_T{}^1(A)$ for look-ahead sets, we merely want to use $F_T{}^k(A) = \{\sigma \in V_T{}^* \mid S \rightarrow{}^* \rho A\beta$ and $\sigma = k{:}\beta$ for some ρ and $\beta\}$; i.e. the set of strings of k or less terminals that may follow the nonterminal A in some sentential form. The intent is that the parsing algorithm would look k symbols ahead, rather than just one, whenever necessary to make a parsing decision. The techniques described below, then, need to be used only for inadequate states with overlapping simple 1-look-ahead sets associated with their terminal- and #-transitions; i.e. only for states for which one-symbol look-ahead is inadequate.

Unfortunately, the following rather long-winded definitions are required to precisely define $SLR(k)$.

Definition. (Recursive on the value of k.) Let G be a CF grammar and k be a positive integer. There is associated with each transition of G's CFSM a *simple k-look-ahead set* which is as follows. For a $\#_{A\rightarrow\omega}$-transition, where $A \rightarrow \omega$ is a production, the set is $F_T{}^k(A)$. For a transition under a symbol s in V, the set is $\{s\}$ if $s \in V_N$ or if $k = 1$, and otherwise is $\{s\beta \in V_T{}^* \mid$ the s-transition is to a state N and β is in a simple $(k-1)$-look-ahead set associated with some terminal- or #-transition from $N\}$.

Although, for ease of definition, sets are associated with every transition of the CFSM, we are interested only in the sets for transitions from inadequate states.

For the value as an example we illustrate the computation of the simple 3-look-ahead set for the $\uparrow$-transition in Figure 2. The computation is actually unnecessary for grammar G_1, since G_1 is $SLR(1)$, as we saw above.

First, we follow all paths leading from state 7, starting with the $\uparrow$-transition, until either a string of length 3 is "spelled out" or until the terminal state is reached. The strings spelled out by all such paths are $\uparrow T\#_{T\rightarrow P}\uparrow T$, $\uparrow P\#_{T\rightarrow P}$, $\uparrow P\uparrow$, $\uparrow i\#_{P\rightarrow i}$, $\uparrow (E$, $\uparrow (T$, $\uparrow (P$, $\uparrow (i$, and $\uparrow (($. Next, the desired set of strings can be derived from these strings as follows. First, each string in $V_T{}^*$ is in the desired set. Second, for each string of the form $\sigma\#_{A\rightarrow\omega}$ (where $A \rightarrow \omega$ is a production, σ is in $V_T{}^*$, and $|\sigma| = n$), every string which can be formed by concatenating σ with a member of $F_T^{k-n}(A)$ is in the desired set. In our special case, the latter means $\uparrow i$ concatenated with the members of $F_T{}^1(P)$. Thus the simple 3-look-ahead set for the $\uparrow$-transition is $\{\uparrow(i, \uparrow((, \uparrow i\uparrow, \uparrow i\dashv, \uparrow i+, \uparrow i)\}$.

Note that if we are parsing a terminal string, no nonterminals can appear in the second or greater positions ahead; however, a nonterminal *can* appear in the first position ahead, due to the last clause of part (3) of our parsing algorithm given above. If we were interested in parsing strings with some nonterminals in them, e.g. in incremental compiling, then strings with nonterminals in them would be considered, too.[2]

Finally we come to our main definition.

Definition. Let k be a positive integer. A CF grammar G is *Simple LR(k)* (abbreviated $SLR(k)$) if and only if for each inadequate state N (if any) of G's CFSM the simple k-look-ahead sets associated with the terminal- and #-transitions from N are mutually disjoint. G is $SLR(0)$ if and only if it is $LR(0)$.

Parsers for $SLR(k)$ grammars are constructed in a manner similar to that described for $SLR(1)$ grammars above, the only differences being that: (1) the look-ahead sets contain strings of length k rather than 1, and (2) when the CFSM is in a look-ahead state, the parsing algorithm must investigate the next k symbols to be read rather than the next one.

It should be clear that the computation of look-ahead sets for a given inadequate state is independent of that for any other state; i.e. we can have a different value of k for each inadequate state. (In a sense, each reduce state has $k = 0$.) Further, the strings in a given look-ahead set L need not all be the same length: each string in L may be of minimum length such that it is not a prefix of some other string in another look-ahead set L_1 of the same inadequate state. Clearly, minimizing the lengths of strings in this manner leaves the look-ahead sets mutually disjoint.

Extension to LR(k) Grammars

At this point we should like to discuss the distinction between $SLR(k)$ and $LR(k)$ grammars. Unfortunately, a thorough discussion is beyond the scope (and the size) of this paper. A thorough discussion is how-

[2] In a paper to follow [5] we describe a generalization of $LR(k)$ in which the parser reduces a string little by little as it scans back and forth over the string. There, too, strings containing nonterminals must be considered.

ever given in [2] and in the forthcoming [4], in which a method is described for extending the above techniques so that a parser can be constructed for any $LR(k)$ grammar.

The method is conceptually simple, although the details get tedious. The method involves computing corresponding pairs of left and right contexts that are pertinent to the decisions that must be made when the CFSM enters an inadequate state, and using these pairs to split states of the CFSM and to compute more restricted look-ahead sets.

In essence, the parsing algorithm uses the CFSM to remember pertinent facts about the history of parses. The state-splitting enhances the memory of the CFSM and thus the capabilities of the parser. The more restricted look-ahead sets are merely subsets of the simple k-look-ahead sets and they contain only right contexts which correspond to specific left contexts remembered by the CFSM.

Our essential contribution vis-a-vis the techniques of Knuth is the decoupling of the computations of look-ahead sets and state-splitting. Korenjak [10] gets a similar effect by partitioning the grammar. In a sense, our technique is Korenjak's taken to the nth degree, i.e. we consider the grammar production by production, look-ahead set by look-ahead set.

In short, then, an $SLR(k)$ grammar is an $LR(k)$ grammar for which simple, straightforward methods suffice for computing look-ahead sets and a minimal CFSM. (The CFSM is minimal in the sense of theorem 3.2 of [8], since it can be shown that it is a *reduced*, deterministic FSM). An $LR(k)$ grammar that is not $SLR(k)$ is one for which more complex techniques are required to compute its minimal parser, but for which the parsing algorithm is the same as for an $SLR(k)$ grammar. That is, the distinction between $SLR(k)$ and $LR(k)$ grammars is a matter of the complexity of the techniques necessary to compute parsers, not a matter of different parsing algorithms.

5. Conclusion

The $SLR(k)$ grammars appear to be an important class of CF grammars. It has been shown in [2] that the $SLR(1)$ grammars include the weak precedence [9] and the simple precedence [14] grammars as proper subsets. Furthermore, the $SLR(k)$ techniques are extendable [2, 4] to cover all $LR[k]$ grammars, i.e. all grammars whose sentences can be parsed during a single deterministic scan from left to right with no more than k symbols of look-ahead.

$SLR(1)$ systems have been implemented at UCSC (University of California at Santa Cruz) [3] and at the University of Toronto [7]. They have been compared with a similar system based on the mixed strategy precedence (MSP) technique of McKeeman [12]. The MSP and $SLR(1)$ systems were used to generate parsers for the languages XPL [12], SPL [15], EULER [14], ALGOL 60 [13], and PAL [16]. Here at UCSC, the $SLR(1)$ system generated parsers in from one-twentieth to one-tenth the time required by the MSP system. The detailed study at Toronto indicates that the resulting $SLR(1)$ parsers require two-thirds or less of both the storage space and the running times of the MSP parsers.

The MSP system failed on the ALGOL 60 (modified slightly to be unambiguous) and the PAL grammars; i.e. these grammars are not MSP. The grammars are, however, $SRL(1)$. The entire $SLR(1)$ parser for ALGOL 60, including the parsing algorithm (with error-recovery), the CFSM, and the look-ahead sets, requires 4237 bytes of storage on an IBM 360. On the order of one minute of CPU time is required to construct these parsers on our IBM 360/40 at UCSC.

Appendix

Definition (restatement of above).

$F_T^1(A) = \{s \in V_T \mid S \rightarrow^* \rho A s \beta \text{ for some } \rho, \beta \in V^*\}$.

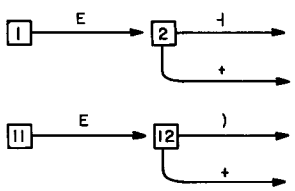
Observation. $S \rightarrow^* \rho A s \beta$ implies either

$S \rightarrow^* \rho' A' \beta' \rightarrow \rho' \rho'' A s \beta'' \beta' = \rho A s \beta$, or
$S \rightarrow^* \rho' A' \beta' \rightarrow \rho' \rho'' A'' s \beta'' \beta' \rightarrow^* \rho' \rho'' \rho''' A s \beta'' \beta' = \rho A s \beta$.

Definition. $[A] = \{B \in V_N \mid B \rightarrow^* \rho' A \text{ for some } \rho' \in V^*\}$.

Theorem. $F_T^1(A) = \{s \in V_T \mid$ *for some* B *in* $[A]$ *there exists a* B*-transition in the CFSM to a state having an* s*-transition*$\}$.

Example. For grammar $G_1[T] = \{T, E\}$, since $T \rightarrow^* T$ by definition and $E \rightarrow^* \rho' T$, in particular, $E \rightarrow T$ and $E \rightarrow E + T$. Thus, $F_T^1(T) = \{\dashv, +,)\}$ since in G_1's CFSM we find



There are no T-transitions to states having transitions under any symbols in V_T.

Proof of Theorem. Given the observation above and the definition of $[A]$, we need only prove that if $A' \rightarrow \rho''Bs\beta''$ is a production, there will be in the CFSM a B-transition to a state having an s-transition. But this follows from the fact that we assumed there were no useless productions and from the method of computing the successors of configuration sets. Thus the configuration $A' \rightarrow \rho'' \cdot Bs\beta''$ must appear in some configuration set which will have a B-successor containing $A' \rightarrow \rho''B \,.\, s\rho''$, and the B-successor will have an s-successor containing $A' \rightarrow \rho''Bs \,.\, \beta''$. Q.E.D.

Conclusion. One can compute $F_T{}^1(A)$ as follows. First, compute $[A]$ directly from the grammar via fast bit matrix techniques [1]. Second, starting with an empty set L, scan the transitions of the CFSM: for each transition under a symbol in $[A]$ to some state N, add to L each symbol s in V_T such that there is an s-transition from N. This step can also be done with bit matrix techniques. The resulting set L is the desired set $F_T{}^1(A)$.

Received May 1970; revised November 1970

References

1. Cheatham, T. E. Jr. *The Theory and Construction of Compilers.* Massachusetts Computer Associates, Inc., Wakefield, Mass., 1967.
2. DeRemer, F. L. Practical translation for $LR(k)$ languages. Ph.D. thesis. MIT, Cambridge, Mass. Sept., 1969.
3. DeRemer, F. L. Simple $LR(k)$ grammars: Definition and implementation. *CEP Rep. 2*, 4 (Sept. 1970), U. of Calif., Santa Cruz.
4. De Remer, F. L. Minimal $LR(k)$ parsers (In preparation).
5. De Remer, F. L. Error recovery using $LR(k)$ techniques (In preparation).
6. Earley, J. An efficient context-free parsing algorithm. *Comm. ACM 13*, 2 (Feb. 1970), 94–102.
7. Horning, J. J., and Lalonde, W. R. Empirical comparison of $LR(k)$ and precedence parsers. Tech. Rep. CSRG-1, Computer Syst. Res. Gr., U. of Toronto, Canada, undated (rec. Sept. 1970).
8. Hopcroft, J. E., and Ullman, J. D. *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, Mass., 1969.
9. Ichbiah, J. D., and Morse, S. P. A technique for generating almost optimal Floyd-Evans productions for precedence grammars. *Comm. ACM 13*, 8 (Aug. 1970), 501–508.
10. Korenjak, A. J. A practical method for constructing $LR(k)$ processors. *Comm. ACM 12*, 11 (Nov. 1969), 613–623.
11. Knuth, D. E. On the translation of languages from left to right. *Inf. Contr. 8* (Oct. 1965), 607–639.
12. McKeeman, W. M., Horning, J. J., and Wortman, D. B. *A Complier Generator.* Prentice-Hall, Englewood Cliffs, N.J., 1970.
13. Naur, P. (Ed.) Revised report on the algorithmic language ALGOL 60. *Comm. ACM 6*, 1 (Jan. 1963), 1–17.
14. Wirth, N. and Weber, H. EULER: A generalization of ALGOL, and its formal definition: I, II. *Comm. ACM 9*, 1, 2 (Jan., Feb., 1966) 13–25, 89–99.
15. Reddy, D. R., and McKeeman, W. M. Computer programming: An introduction to PL. Prelimin. Rep., Stanford U., Stanford, Calif., 1969.
16. Evans, A. Jr. PAL—a language designed for teaching programming linguistics. Proc. ACM 23rd Nat. Conf., 1968, Brandon/Systems Press, Princeton, N.J., pp. 395–403.

A Language Extension for Graph Processing and Its Formal Semantics

Terrence W. Pratt and Daniel P. Friedman
The University of Texas at Austin*

A simple programming language "extension," Graspe, for processing directed graphs is defined. Graspe consists of a type of directed graph data structure and a set of primitive operations for manipulating these structures. Graspe may be most easily implemented by embedding it in a host language. Emphasis is placed both on Graspe itself and on its method of definition. Commonly, the definition of a language involves definition of the syntactic elements and explanation of the meaning to be assigned them (the semantics). The definition of Graspe here is solely in terms of its semantics; that is, the data structures and operations are defined precisely but without assignment of a particular syntactic representation. Only when the language is implemented is assignment of an explicit syntax necessary. An example of an implementation of Graspe embedded in Lisp is given as an illustration. The advantages and disadvantages of the definition of a language in terms of its semantics are discussed.

Key Words and Phrases: graph processing, programming language, formal semantics, directed graph, Lisp, network, data structure, flowchart, syntax, language definition

CR Categories: 4.20, 4.22, 5.23, 5.24, 5.32

* Department of Computer Science. This work was supported in part by NSF Grant GJ-778.