Simple LR(k) Grammars:

Definition and Implementation

FRANKLIN L. DE REMER

CEP REPORT, VOLUME 2, No. 4

September 1970

The Computer Evolution Project
Applied Sciences
The University of California at
Santa Cruz, California 95060

Simple LR(k) Grammars:

Definition and Implementation

ABSTRACT

We define herein a class of context-free grammars, called the "Simple LR(k)" or "SLR(k)" grammars, which have been shown to include the weak precedence and the simple precedence grammars as proper subsets. We also show how to construct parsers for the SLR(k) grammars. Our parser constructing techniques have been implemented (see Appendices II and III) and by direct comparison have been shown 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.

TABLE OF CONTENTS

Simple LR(k) Grammars: Definition and Implementation

I. Introduction

This paper reports some of the results attained in the course of thesis research[+] by the author. We define a class of context-free grammars, called the "Simple LR(k)" or "SLR(k)" grammars, which have been shown (DeR 69) to include the weak precedence (I&M 70) and the simple precedence (W&W 66) grammars. We also show how to construct parsers for SLR(k) grammars. The construction technique can be extended (DeR 69, 70a) to an algorithm for constructing a parser for any LR(k) grammar (Knu 65); i.e., any grammar which generates strings each of which can be parsed during a single deterministic scan from left to right without looking ahead more than k symbols.

We also present in Appendix II an implementation of our SLR(k) techniques[++]. The presentation is in the form of a

---

[+]The thesis research was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102 (01).

[++]The implementation was supported in part by NASA under grant number NGR05-061-005.

debugged, operating program written in the language XPL (MHW 70). The program analyses grammars and, if they are SLR(1), it punches tables which can be plugged into the "skeleton" given in Appendix III to form syntax analysers for the grammars. The syntax analysers can then be used in syntax-directed compilers for languages based on these grammars (see MHW 70).

II. 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, lower case Latin letters and special symbols (e.g., +, *, :, etc.) to denote terminals, and lower case 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:

$$S \rightarrow \vdash E \dashv \qquad T \rightarrow P \uparrow T \qquad P \rightarrow i$$

$$E \rightarrow E + T \qquad T \rightarrow P \qquad P \rightarrow (E)$$

$$E \rightarrow T$$

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 $\alpha' \overset{*}{\rightarrow} \alpha$, which means there exist strings $\alpha_0, \alpha_1, \ldots, \alpha_n$ such that $\alpha' = \alpha_0 \rightarrow \alpha_1 \rightarrow \ldots \rightarrow \alpha_n = \alpha$ for $n \geq 0$. 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., $L(G) = \{\eta \in V_T^* \mid S \overset{*}{\rightarrow} \eta\}$. We assume that G has no useless productions; i.e., we assume that for each production $A \rightarrow \omega$ there exists a derivation $S \overset{*}{\rightarrow} \sigma\ A\ \beta \rightarrow \sigma\ \omega\ \beta \overset{*}{\rightarrow} \sigma\ \delta\ \beta$ where σ, δ, and β are terminal strings. Well known methods exist for detecting and removing useless productions (H&U 69).

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.

III. Parsers for LR(0) Grammars

In this section we briefly review the results of Knuth (Knu 65) regarding LR(0) grammars. We use a combination of the terminologies of Earley (Ear 70) and McKeeman (McK 70).

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.

Each non-empty configuration set has one or more successors, other configuration sets. For instance, S_0 has a single successor (in particular, 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 such that there exists a configuration in S_i with a marker preceding an instance of the symbol s.

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.

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.

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.

We can abstract from a set of configuration sets and their successor relations 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 (DeR 69) 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 but one transition, and it under one of the special $\#_{A \rightarrow w}$-symbols, is called a reduce state. States having both kinds of transitions 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).) Also, he showed that the following parsing algorithm is correct for an LR(0) grammar G.

State Names (Numbers)	Configuration Sets	Successor Relations
0:	{S → . ⊢ E ⊣ }	$\xrightarrow{\vdash}$ 1
1:	{S → ⊢ . E ⊣ E → . E + T E → . T T → . P ↑ T T → . P P → . i P → .(E) }	$\xrightarrow{E}$ 2 $\xrightarrow{T}$ 6 $\xrightarrow{P}$ 7 $\xrightarrow{i}$ 10 $\xrightarrow{(}$ 11
2:	{S → ⊢ E . ⊣ E → E . + T }	$\xrightarrow{\dashv}$ 3 $\xrightarrow{+}$ 4
3:	{S → ⊢ E ⊣ . }	$\xrightarrow{\#S \to \vdash E \dashv}$ 14
4:	{E → E + . T T → . P ↑ T T → . P P → . i P → . (E) }	$\xrightarrow{T}$ 5 $\xrightarrow{P}$ 7 $\xrightarrow{i}$ 10 $\xrightarrow{(}$ 11
5:	{E → E + T. }	$\xrightarrow{\#E \to E+T}$ 14
6:	{E → T.}	$\xrightarrow{\#E \to T}$ 14
7:	{T → P.↑ T T → P. }	$\xrightarrow{\uparrow}$ 8 $\xrightarrow{\#T \to P}$ 14
8:	{T → P ↑. T T → .P ↑ T T → .P P → .i P → .(E) }	$\xrightarrow{T}$ 9 $\xrightarrow{P}$ 7 $\xrightarrow{i}$ 10 $\xrightarrow{(}$ 11
9:	{T → P ↑ T. }	$\xrightarrow{\#T \to P \uparrow T}$ 14
10:	{P → i.}	$\xrightarrow{\#P \to i}$ 14
11:	{P → (.E) E → .E + T E → .T T → .P ↑ T T → .P P → .i P → .(E) }	$\xrightarrow{E}$ 12 $\xrightarrow{T}$ 6 $\xrightarrow{P}$ 7 $\xrightarrow{i}$ 10 $\xrightarrow{(}$ 11
12:	{P → (E.) E → E. + T }	$\xrightarrow{)}$ 13 $\xrightarrow{+}$ 4
13:	{P → (E) . }	$\xrightarrow{\#P \to (E)}$ 14
14:	{ }	

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

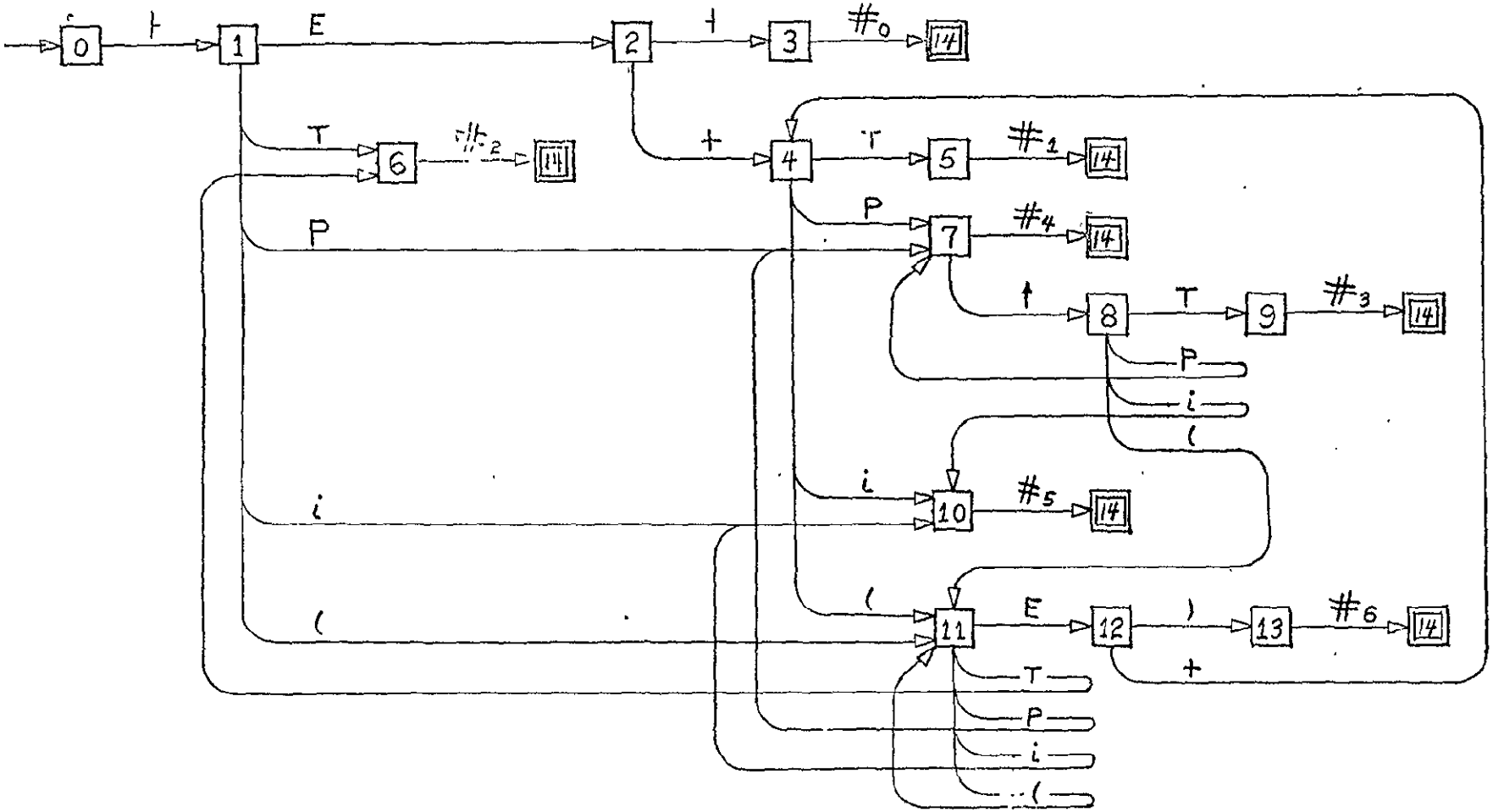


Figure 2. The CFSM of our example grammar G_1: (0) S → ⊢ E ⊣, (1) E → E + T, (2) E → T, (3) T → P ↑ T, (4) T → P, (5) P → i, (6) P → (E). 14 denotes a single state, the final state. The subscripts on the #-symbols reference productions.

LR(0) parsing algorithm: Maintain a stack on which to store alternately symbols in V and the names of 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):

(i) Let $\alpha = \eta$.

(ii) 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.

(iii) 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 transition from 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 the top of the stack, set $\alpha = A\alpha$, and go to step (ii).

It has been shown in (DeR 69) and in (Knu 65) 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 $L(G_1)$

TABLE I.

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

Stack	Input	Output	Line #
0	$\vdash i + i \dashv$		1
$0\vdash 1$	$i + i \dashv$		2
$0\vdash 1 i_{10}$	$+ i \dashv$		3
$0\vdash 1 P_7$	$+ i \dashv$	$P \to i$	4†
$0\vdash 1 T_6$	$+ i \dashv$	$T \to P$	5
$0\vdash 1 E_2$	$+ i \dashv$	$E \to T$	6
$0\vdash 1 E_2 +_4$	$i \dashv$		7
$0\vdash 1 E_2 +_4 i_{10}$	$\dashv$		8
$0\vdash 1 E_2 +_4 P_7$	$\dashv$	$P \to i$	9†
$0\vdash 1 E_2 +_4 T_5$	$\dashv$	$T \to P$	10
$0\vdash 1 E_2$	$\dashv$	$E \to E + T$	11
$0\vdash 1 E_2 \dashv_3$			12
0		$S \to \vdash E \dashv$	fini

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

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 → . ⊢ E ⊣ }, indicating that we are expecting an instance of an S which is composed of a ⊢ followed by an E followed by a ⊣. The marker "." before the ⊢ indicates that the next thing to be matched is a ⊢.

If ⊢ is the next symbol in the string, we read it and proceed to a state represented by {S → ⊢ . E ⊣, E → . E + T, E → . T, T → . P ↑ T, T → . P, P → . i , P → . (E) }; otherwise the string is in error (not in L(G)). The latter set indicates that the next symbol must be an E, which may be in the form of an E + T or a T, and the T may be in the form P ↑ 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 → . i. Thus, we read the i and proceed to a state of the parse represented by {P → 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.

IV. 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).

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 $T \rightarrow P \uparrow T$

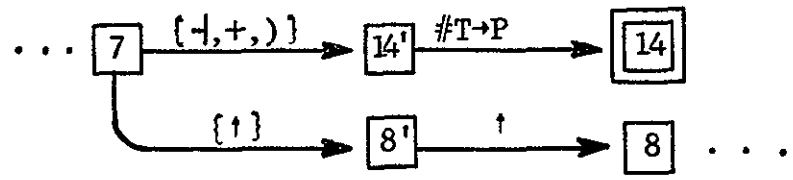
implies that ↑ is in $F_T^1(P)$. The production T → P implies that all the strings in $F_T^1(T)$ are also in $F_T^1(P)$. E → E + T and E → T each imply that the members of $F_T^1(E)$ are also in $F_T^1(T)$. S → ⊢ E ⊣ implies that ⊣ is in $F_T^1(E)$; E → E + T adds +; and P → (E) adds ")". Thus, we have determined that $F_T^1(P) = \{\uparrow, \dashv, +,) \}$, and in the process that $F_T^1(T) = F_T^1(E) = \{ \dashv, +,) \}$. In Appendix I 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: {↑} for the ↑-transition and $F_T^1(T) = \{ \dashv, +,) \}$ for the $\#_{T \to P}$-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.

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 #
7	$\vdash_0 {}_1 P_7$	+ i ⊣		4
14'	$\vdash_0 {}_1 P_7$	+ i ⊣	T → P	4'
6	$\vdash_0 {}_1 T_6$	+ i ⊣		5

Generalization to SLR(k). Conceptually, the generalization from SLR(1) to SLR(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 $\#_{A \to \omega}$-transition, where $A \to \omega$ 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 $\{s\beta \in V_T^* \mid$ 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 ↑-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 ↑-transition, until either a string of length three is "spelled out" or until the terminal state is reached. The strings spelled out by all such paths are $\uparrow T\#_{T\to P\uparrow T}$, $\uparrow P\#_{T\to P}$, $\uparrow P\uparrow$, $\uparrow i\#_{P\to i}$, ↑(E, ↑(T, ↑(P, ↑(i, and ↑((. 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 $\sigma\#_{A\to\omega}$, where $A \to \omega$ 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 ↑i concatenated with the members of $F_T^1(P)$. Thus, the simple 3-look-ahead set for the ↑-transition is { ↑(i, ↑((, ↑i↑, ↑i┤, ↑i+, ↑i) }.

Note that if we are parsing a terminal string, no nonterminals can appear in the second or greater positions ahead; however, a nonterminal <u>can</u> appear in the first position ahead, due to the last clause of part (iii) 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[+].

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 (1) that the look-ahead sets contain strings of length k rather than one, and (2) that 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

---

[+]In a paper to follow (DeR 70b) 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.

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. In (DeR 69) and in a forthcoming paper (DeR 70a) a method is described for extending the above techniques so that a parser can be constructed for any LR(k) grammar. Conceptually the method is simple, although the details get tedious. Basically 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.

V. Conclusion.

The SLR(k) grammars appear to be an important class of CF grammars. It has been shown in (DeR 69) that the SLR(1) grammars include the weak precedence (I&M 70) and the simple precedence (W&W 66) grammars as proper subsets. Furthermore, the SLR(k) techniques are extendable (DeR 69, 70a) first to cover all "right bounded context" grammars (Flo 64), and then, with a bit more work, to cover all LR(k) grammars; i.e., grammars whose sentences can be parsed during a single deterministic scan from left to right with no more than k symbols of look-ahead.

An SLR(1) system has been implemented here at UCSC (see Appendix II) and at the University of Toronto[+], and they have been compared with a similar system based on the extended precedence methods of McKeeman (McK 66, MHW 70). The precedence and SLR(1) systems were used to generate parsers for the languages XPL (MHW 70), SPL (M&R 69), and PAL (Eva 69). The SLR(1) system generated parsers for XPL and SPL in about one fifth the time required by the extended precedence system; the parsers generated by the two systems require comparable space and running times. An SLR(1) parser was generated for PAL, but an extended precedence parser could not be constructed for it without modifying the grammar; i.e., PAL's grammar is not extended precedence.

[+]Personal communication from Professor James J. Horning, Computer Systems Research Group, University of Toronto, Toronto Canada, 6 August 1970.

APPENDIX I

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

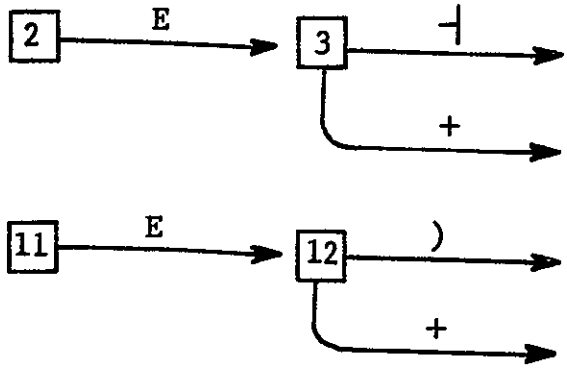
$S \rightarrow^* \rho'\, A'\, \beta' \rightarrow \rho'\, \rho''\, A\, s\, \beta''\, \beta' = \rho\, A\, s\, \beta$ or

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

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, $F_T^1(T)$ = {⊣, +,)} since in G_1's CFSM we find



Several minor errors have been detected in the following program since publication. Please write for a revised sheet if the portions are of interest to you.

APPENDIX II

X P L COMPILATION - U OF C AT SANTA CRUZ - XCOM III VERSION OF APRIL 27, 1970. CLOCK

TODAY IS SEPTEMBER 23, 1970. CLOCK TIME = 11:3:33.65.

```
/*****************************************************************************
*                                                                           *
*                                                                           *
*                        A SIMPLE LR(K) GRAMMAR ANALYSER                    *
*                                                                           *
*                                    BY                                     *
*                                                                           *
*                              FRANKLIN L. DE REMER                         *
*                                                                           *
*                            UNIVERSITY OF CALIFORNIA                       *
*                            SANTA CRUZ, CALIFORNIA                         *
*                                AUGUST 31, 1970                            *
*                                                                           *
*                                                                           *
*                                  PREFACE                                  *
*                                                                           *
*     THE FOLLOWING PROGRAM IS AN IMPLEMENTATION OF THE SIMPLEST RESULTS    *
*PRESENTED IN THE AUTHOR'S DISSERTATION (DER 69).  THE PROGRAM READS A GRAMMAR*
*AND ATTEMPTS TO CONSTRUCT A SIMPLE LR(1) PARSER FOR IT.  THE PROGRAM EITHER *
*SUCCEEDS AND PUNCHES TABLES THAT REPRESENT THE PARSER, OR IT PRINTS MESSAGES*
*RELATED TO THE REASONS WHY IT CANNOT BE SUCCESSFUL.  IF THE PROGRAM FAILS,  *
*THE GIVEN GRAMMAR IS EITHER TOO LARGE IN SOME ASPECT OR IT IS NOT SIMPLE LR(1):
*I.E., THE PROGRAM WILL SUCCEED, EXCEPT FOR SPACE LIMITATIONS, FOR ANY SIMPLE *
*LR(1) GRAMMAR.                                                             *
*     IT IS TO BE EMPHASIZED THAT THIS PROGRAM IS INTENEDED MORE AS DOCUMENTATION
*THAN AS THE MOST EFFICIENT IMPLEMENTATION OF SLR(1) TECHNIQUES.  THUS, BY AND *
*LARGE, WE OPTED FOR READABILITY RATHER THAN EFFICIENCY WHEN THE TWO WERE IN  *
*CONFLICT.  FURTHERMORE, THE TABLES GENERATED BY THE PROGRAM REPRESENT A PARSER *
*WHICH WILL EXECUTE QUICKLY BUT WHICH IS QUITE SPACE-INEFFICIENT.  THE SIZE   *
*OF THE TABLES GENERATED COULD BE REDUCED TO ABOUT ONE QUARTER THE CURRENT SIZE *
*BY SEPARATING TRANSITIONS UNDER TERMINALS FROM THOSE UNDER NONTERMINALS,     *
*COMBINING EQUIVALETT ROWS IN THE RESULTING MATRICES, AND PERHAPS IMPLEMENTING *
*STATES WITH ONLY A FEW TRANSITIONS VIA LISTS RATHER THAN MATRICES.          *
*                                                                           *
*                                                                           *
*                          HOW TO USE THE ANALYSER                          *
*                                                                           *
*     INPUT TO THE PROGRAM IS MADE VIA CARDS.  CARDS WITH THE CHARACTER "@" IN *
*THE FIRST COLUMN ARE TREATED AS COMMENT OR CONTROL CARDS, AS DESCRIBED AT THE *
*BEGINNING OF THE PROGRAM.  BLANK CARDS ARE IGNORED.  GRAMMARS MAY BE BATCHED BY *
*SEPARATING THEM WITH CONTROL CARDS BEGINNING WITH "@EOG".  DO NOT PUT SUCH A *
*CARD AFTER THE LAST GRAMMAR.                                               *
*     PRODUCTIONS ARE PLACED ONE TO A CARD.  IF THE FIRST COLUMN IS NON-BLANK, *
*THE FIRST TOKEN ON THE CARD IS TAKEN TO BE THE LEFT PART OF THE PRODUCTION:  *
*OTHERWISE, THE LEFT PART IS TAKEN TO BE THE LEFT PART OF THE PRECEDING PRODUC-
*TION.  THE BALANCE OF THE CARD IS TAKEN TO BE THE RIGHT PART OF THE PRODUCTION.
*ANY TOKEN THAT DOES NOT OCCUR AS A LEFT PART IS A TERMINAL SYMBOL:  ANY TOKEN *
*THAT OCCURS ONLY AS A LEFT PART IS A GOAL SYMBOL (THERE SHOULD BE AT MOST ONE *
*PER GRAMMAR).  ALL PRODUCTIONS WITH THE SAME LEFT PART MUST BE GROUPED.     *
*     A TOKEN IS EITHER                                                     *
*(1)  THE CHARACTER "<" FOLLOWED BY A BLANK, OR                             *
*(2)  ANY CONSECUTIVE GROUP OF NON-BLANK CHARACTERS NOT BEGINNING WITH "<" AND *
*     FOLLOWED BY A BLANK OR THE END OF THE CARD, OR                        *
*(3)  THE CHARACTER "<" FOLLOWED BY A NON-BLANK CHARACTER AND THEN ANY STRING OF *
```

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 $A' \rightarrow \rho'' B s \beta''$ 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 $A' \rightarrow \rho''.B\ s\ \beta''$ 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 $A' \rightarrow \rho''\ B\ s\ .\ \beta''$. 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 (Che 67). 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 (see Appendix II). The resulting set L is the desired set $F_T^1(A)$.

```
    |     BLANK AND NON-BLANK CHARACTERS UP TO AND INCLUDING THE NEXT OCCURRENCE OF
    |     THE CHARACTER ">".
    |
    |
    |                                ACKNOWLEDGEMENTS
    |
    |     THE GRAMMAR REPRESENTATION AND THE PROCEDURES, ERROR, LINE_OUT, PRINT_DATE,
    |PRINT_TIME, PRINT_G, READ_G, AND PUNCH_CARD, WERE ADAPTED FROM SIMILAR
    |PROCEDURES IN THE XPL GRAMMAR ANALYSER (MHW70) AND WERE ORIGINALLY WRITTEN
    |BY J. J. HORNING UNDER THE SUPERVISION OF W. M. MCKEEMAN.  THESE PROCEDURES
    |WERE, HOWEVER, REFORMATTED, RESCOPED, AND REPARAGRAPHED ACCORDING TO OUR OWN
    |CONVENTIONS.
    |     WE WOULD ALSO LIKE TO THANK JEFFREY SUE FOR PRELIMINARY WORK ON AND
    |DISSCUSSIONS ABOUT AN SLR(1) ANALYSER AND W. M. MCKEEMAN FOR NUMEROUS
    |SUGGESTIONS REGARDING THE IMPLEMENTATION.
    |
    |
    |                                     INDEX
    |
    |     THE GROSS STRUCTURE OF THE PROGRAM, ALONG WITH LINE NUMBERS AND SOME
    |COMMENTS, IS GIVEN BELOW:
    |
    |LINE
    |NO.   SUBJECT
    |
    | 170  LANGUAGE EXTENTIONS (LITERAL DECLARATIONS):
    |         BOOLEAN, TRUE, FALSE, PUNCH, EJECT_PAGE, DOUBLE_SPACE,
    |         AT, BLANK, BLANK_CARD, HALF_LINE, X12.
    |
    | 190  SYSTEM TOGGLES:
    |         MORE_GRAMMARS, CONTROL, @I, @G, @C, @F,
    |         @L, @P.
    |
    | 218  UTILITY PROCEDURES: ODD, MIN, MAX, ERROR(ERROR_COUNT),
    |         LINE_OUT, PRINT_DATE, PRINT_TIME (FIRST_TIME,
    |         THIS_TIME, LAST_TIME).
    |
    | 293  GRAMMAR REPRESENTATION: PROD_ARRAY, PROD_ARRAY-PTR, PROD_START,
    |         RT_PT_SIZE, FIRST_PROD_FOR, ON_LEFT, ON_RIGHT,
    |         V, #_PRODS, #_TERMINALS, #_NTS, GOAL_SYMBOL,
    |         IS_NONTERMINAL, LEFT_PART.
    |
    | 323  PRINT_G:  /* A PRODECURE TO PRINT A GRAMMAR IN STANDARD FORMAT.  * /
    |
    | 402  READ_G:  /* A PROCEDURE TO READ A GRAMMAR.  * /
    |         FIND_GOAL:
    |         SORT_V:
    |         SCAN:
    |         GETTING_CARDS:
    |         /* THE BODY OF READ_G.  * /
    |
    | 626  CFSM REPRESENTATION: READ_XITIONS_TABLE,
    |         SYM_BEFORE, #_TO_POP, SYM_TO_READ,
    |         DEFAULT_XITION, IS_INADEQUATE, IS_REDUCE,
    |         IS_LR0, IS_SLR1, EXIT_STATE, ERROR_STATE,
    |         IS_LA_REDUCE, IS_READ, #_LA_REDUCE_STATES,
    |         INC_#_LA_REDUCE_STATES, #_READ_STATES,
    |         INC_#_READ_STATES, READ_XITION, UPDATE_READ_XITION,
    |         FLAG, UNFLAG, I_TO_S4.
    |
    | 695  PRINT_CFSM:  /* A PROCEDURE TO PRINT A CFSM.  * /
```

```
116 |          STR_LNTH_8:
117 |          /* THE BODY OF PRINT_CFSM.  * /
118 |
119 | 812  COMPUTE_CFSM:  /* A PROCEDURE TO COMPUTE A CFSM.  * /
120 |             MAKE_CONFIG:
121 |             PROD:
122 |             DOT:
123 |             SYM_AFTER_DOT:
124 |             DOT_AT_END:
125 |             SUCCESSOR:
126 |             (BASIS_STACK, BS_START, BS_SIZE)
127 |             (CONFIG_SET, #_CONFIGS, INC_#_CONFIGS)
128 | 869      COMPUTE_CONFIG_SET:
129 |             (STATE)
130 |             PRINT_CS:
131 |             ADD_SUCCS_TO_CS:
132 |             SORT_CS:
133 |             /*THE BODY OF COMPUTE_CONFIG_SET.  * /
134 |          (READ_TO_INAD, STATE_PTRS)
135 | 959      ADD_SUCCESSORS_TO_BS:
136 |             (STATE, BASIS_SET, INC_BASIS_SET_SIZE)
137 |             LOOK_UP_READ:
138 |             LOOK_UP_REDUCE:
139 |             /* THE BODY OF ADD_SUCCESSORS_TO_BS.  * /
140 |1065      /* THE BODY OF COMPUTE_CFSM.  * /
141 |
142 |1117  SLR(1) LOOK-AHEAD SET REPRESENTATION:
143 |          (LOOK_AHEAD_TABLE)
144 |          VALIDATE_LA_XITION:
145 |          IS_VALID_LA:
146 |          INCLUDED:
147 |          INCLUDE:
148 |          INTERSECT:
149 |
150 |1185  PRINT_SLR1_LA_SETS:  /* A PROCEDURE TO PRINT SLR(1) LOOK-AHEAD SETS.  * /
151 |          STR_LNTH13:
152 |          /* THE BODY OF PRINT_LOOK_AHEAD_SETS.  * /
153 |
154 |1228  COMPUTE_SLR1_LA_SETS:  /*A PROCEDURE TO COMPUTE SLR(1) LOOK-AHEAD SETS.* /
155 |          NOT_SLR1:
156 |          /* THE BODY OF COMPUTE_SLR1_LA_SETS.  * /
157 |
158 |1314  PUNCH_DPDA:  /* A PROCEDURE TO PUNCH SLR(1) PARSERS.  * /
159 |          PUNCH_CARD:
160 |          CONVERT:
161 |          /* THE BODY OF PUNCH_DPDA. * /
162 |
163 |1465  MAIN PROGRAM.
164 |1531  END OF PROGRAM.
165 |
166 | ************************************************************************/
167 |
168 |
169 |
170 | /*FIRST SCME LANGUAGE EXTENTIONS.  ********************************************/
171 |
172 |   DECLARE BOOLEAN      LITERALLY 'BIT(1)',
173 |           TRUE         LITERALLY '1',
174 |           FALSE        LITERALLY '0',
175 |           EJECT_PAGE   LITERALLY 'OUTPUT(1) = ''1''',
176 |           DOUBLE_SPACE LITERALLY 'OUTPUT(1) = ''0''',
```

```
177 |           PUNCH        LITERALLY '2',
178 |           BLANK        LITERALLY 'BYTE('' '')',
179 |           AT           LITERALLY 'BYTE(''@'')',
180 |           BLANK_CARD   LITERALLY '''
181 |                                  ''',    /*  IMAGE OF A BLANK CARD.  */
182 |           HALF_LINE    LITERALLY '''
183 |                         ''',
184 |           DASHED_LINE  LITERALLY '''          ----------
185 |----------------------------------------------------------------------
186 |---------''',
187 |           X12          LITERALLY '''            ''';    /*  TWELVE BLANKS.  */
188 |
189 |
190 | /*NEXT SOME SYSTEM TOGGLES.  ************************************************/
191 |
192 |   DECLARE MORE_GRAMMARS BOOLEAN;
193 |     /*  CONTROLS THE BATCHING OF GRAMMARS.  */
194 |
195 |   DECLARE CONTROL(255) BOOLEAN,
196 |     /*  A @ IN COLUMN ONE OF A GRAMMAR CARD SWITCHES THE TOGGLE              */
197 |     /*  INDICATED BY THE CHARACTER IMMEDIATELY FOLLOWING THE @.              */
198 |
199 |         @I LITERALLY 'BYTE(''I'')',
200 |     /*  @I CONTROLS LISTING OF INPUT GRAMMAR CARDS;  INITIALLY OFF.          */
201 |
202 |         @G LITERALLY 'BYTE(''G'')',
203 |     /*  @G CONTROLS LISTING OF REFORMATTED GRAMMAR; INITIALLY ON.           */
204 |
205 |         @C LITERALLY 'BYTE(''C'')',
206 |     /*  @C CONTROLS LISTING OF CONFIGURATION SETS; INITIALLY OFF.           */
207 |
208 |         @F LITERALLY 'BYTE(''F'')',
209 |     /*  @F CONTROLS LISTING OF CFSM;  INITIALLY ON.                         */
210 |
211 |         @L LITERALLY 'BYTE(''L'')',
212 |     /*  @L CONTROLS LISTING OF LOOK-AHEAD SETS;  INITIALLY ON.              */
213 |
214 |         @P LITERALLY 'BYTE(''P'')';
215 |     /*  @P CONTROLS PUNCHING OF DPDA;  INITIALLY ON.                        */
216 |
217 |
218 | /*NOW SOME UTILITY PROCEDURES.  **********************************************/
219 |
220 |   /*
221 |ODD:  */    /*  THE PREDICATE ODD.  */
222 |   DECLARE ODD LITERALLY '';
223 |
224 |MIN:
225 |   PROCEDURE (I, J) FIXED;
226 |     DECLARE (I, J) FIXED;
227 |     IF I < J THEN RETURN I;   ELSE RETURN J;
228 |   END MIN;
229 |
230 |MAX:
231 |   PROCEDURE (I, J) FIXED;
232 |     DECLARE (I, J) FIXED;
233 |     IF I > J THEN RETURN I;   ELSE RETURN J;
234 |   END MAX;
235 |
236 |   DECLARE ERROR_COUNT FIXED;                          /*  GLOBAL VARIABLE.  */
237 |ERROR:              /*  PRINT AN ERROR MESSAGE, INCREMENT ERROR_COUNT.  */
```

```
   PROCEDURE (MESSAGE);
      DECLARE MESSAGE CHARACTER;
      OUTPUT = '*** ERROR, ' || MESSAGE;   DOUBLE_SPACE;
      ERROR_COUNT = ERROR_COUNT + 1;
   END ERROR;

LINE_OUT:            /*  NUMBER A LINE AND PRINT IT.  */
   PROCEDURE (NUMBER, LINE);
      DECLARE NUMBER FIXED, LINE CHARACTER;
      DECLARE NUM FIXED, N CHARACTER;
      N = NUMBER;     /*  CONVERT N TO CHARACTER.  */
      NUM = 6 - LENGTH(N);     /*  6 = MARGIN.  */
      OUTPUT = SUBSTR(X12, 0, NUM) || N || '    ' || LINE;
   END LINE_OUT;

PRINT_DATE:       /*  PRINTS A MESSAGE FOLLOWED BY A DATE.  */
   PROCEDURE (MESSAGE, DATE);
      DECLARE MESSAGE CHARACTER, DATE FIXED;
      DECLARE MONTH(11) CHARACTER INITIAL
             ('JANUARY', 'FEBRUARY', 'MARCH', 'APRIL', 'MAY', 'JUNE',
              'JULY', 'AUGUST', 'SEPTEMBER', 'OCTOBER', 'NOVEMBER', 'DECEMBER'),
              DAYS(11) FIXED INITIAL
             (0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335);
      DECLARE (YEAR, DAY, M) FIXED;
      DAY = DATE MOD 1000;
      YEAR = DATE/1000 + 1900;
      IF (YEAR & 3) ¬= 0 THEN IF DAY > 59 THEN DAY = DAY + 1;
      M = 11;
      DO WHILE DAY <= DAYS(M);  M = M - 1;  END;
      OUTPUT = MESSAGE || MONTH(M) || ' ' || DAY-DAYS(M) || ', ' ||
               YEAR || '.';   DOUBLE_SPACE;
   END PRINT_DATE;

   DECLARE (FIRST_TIME, THIS_TIME, LAST_TIME) FIXED;   /*  GLOBAL VARIABLES.  */
PRINT_TIME:          /*  OUTPUT ELAPSED TIMES.  */
   PROCEDURE;
      DECLARE (I, J) FIXED, T CHARACTER;
      THIS_TIME = TIME;
      I = THIS_TIME - LAST_TIME;
      J = I MOD 100;
      I = I / 100;
      T = 'TIME USED SINCE THE LAST TIME PRINTOUT IS ' || I || '.';
      IF J < 10 THEN T = T || '0';
      OUTPUT = T || J || ' SECONDS.';
      I = THIS_TIME - FIRST_TIME;
      J = I MOD 100;
      I = I / 100;
      T = 'TOTAL TIME USED SO FAR IS ' || I || '.';
      IF J < 10 THEN T = T || '0';
      OUTPUT = T || J || ' SECONDS.';   DOUBLE_SPACE;
      LAST_TIME = THIS_TIME;
   END PRINT_TIME;    /*************************************************************/


/*NEXT THE REPRESENTATION OF A GRAMMAR.  ************************************/

   DECLARE PROD_ARRAY          (2047) BIT(8),
           PROD_ARRAY_PTR             FIXED,
           PROD_START          (255)  BIT(16),
           RT_PT_SIZE          (255)  BIT(8),
```

```
299 |           FIRST_PROD_FOR    (255)  BIT(16),
300 |          (ON_LEFT, ON_RIGHT)(255)  BOOLEAN,
301 |           V                 (255)  CHARACTER,
302 |           (#_PRODS, #_TERMINALS, #_NTS, GOAL_SYMBOL) FIXED,
303 |           INC_PROD_ARRAY_PTR LITERALLY
304 |           'IF PROD_ARRAY_PTR < 2047 THEN PROD_ARRAY_PTR = PROD_ARRAY_PTR + 1;
305 |            ELSE DO; CALL ERROR(''THE GRAMMAR IS TO LARGE.'');
306 |                     PROD_ARRAY_PTR = 2037; END',
307 |           INC_#_PRODS LITERALLY
308 |           'IF #_PRODS < 255 THEN #_PRODS = #_PRODS + 1;
309 |            ELSE CALL ERROR(''TOO MANY PRODUCTIONS.'')';
310 |
311 |      /*
312 |   IS_NONTERMINAL:  */
313 |      DECLARE IS_NONTERMINAL LITERALLY 'ON_LEFT';
314 |
315 |      /*
316 |   LEFT_PART:  */
317 |      DECLARE LEFT_PART LITERALLY 'PROD_ARRAY(PROD_START';
318 |
319 |
320 |
321 | /*THEN A PROCEDURE TO PRINT A GRAMMAR IN STANDARD FORMAT.  ********************/
322 |
323 |PRINT_G:
324 |   PROCEDURE;
325 |
326 |   /* PRINT THE VOCABULARY.  */
327 |
328 |      OUTPUT =
329 |      '                                          T H E   V O C A B U L A R Y';
330 |      DOUBLE_SPACE;
331 |      OUTPUT = '          T E R M I N A L   S Y M B O L S                    '
332 |                                || '              N O N T E R M I N A L S';
333 |      DOUBLE_SPACE;
334 |
335 |      DECLARE N FIXED, LINE CHARACTER;
336 |      DO N = 1 TO MAX(#_TERMINALS, #_NTS);    /*  PRINT THE VOCABULARY.  */
337 |         IF N <= #_TERMINALS     /*  TERMINAL SYMBOLS.  */
338 |            THEN LINE = SUBSTR(V(N) || HALF_LINE, 0, 66);
339 |            ELSE LINE = HALF_LINE;
340 |         IF N <= #_NTS THEN      /*  NONTERMINAL SYMBOLS.  */
341 |            LINE = LINE || V(N + #_TERMINALS);
342 |         CALL LINE_OUT(N, LINE);
343 |      END;
344 |      DOUBLE_SPACE;
345 |
346 |      OUTPUT =
347 |            '          ' || HALF_LINE || V(GOAL_SYMBOL) ||' IS THE GOAL SYMBOL.';
348 |      IF MAX(#_TERMINALS, #_NTS) > 20 THEN EJECT_PAGE;   ELSE DOUBLE_SPACE;
349 |
350 |   /* PRINT THE PRODUCTIONS.  */
351 |
352 |      OUTPUT = '                        T H E   P R O D U C T I O N S';
353 |      DOUBLE_SPACE:          /*  THE PRODUCTION ADDED AUTOMATICALLY   */
354 |      DO N = 1 TO #_PRODS - 1;                  /*   IS NOT PRINTED.   */
355 |         DECLARE SYM FIXED;
356 |         SYM = PROD_ARRAY(PROD_START(N));    /*  SET UP THE LEFT PART.  */
357 |         IF SYM = PROD_ARRAY(PROD_START(N - 1))
358 |         THEN                                                          DO;
359 |            DECLARE L FIXED;
```

```
360 |          L = LENGTH(V(SYM));
361 |          LINE = SUBSTR(HALF_LINE, 0, L) || '    |  ';                    END;
362 |       ELSE                                                               DO;
363 |          OUTPUT = '';
364 |          LINE = V(SYM) || '  ::=  ';                                     END;
365 |
366 |       DECLARE (S, VAR) FIXED;              /*  ADD ON THE RIGHT PART.  */
367 |       VAR = PROD_START(N);
368 |       DO S = VAR + 1 TO VAR + RT_PT_SIZE(N);
369 |          LINE = LINE || ' ' || V(PROD_ARRAY(S));
370 |       END;
371 |       CALL LINE_OUT(N, LINE);
372 |    END;
373 |    EJECT_PAGE;
374 |
375 |  /* PRINT STATISTICS ON THE GRAMMAR.  */
376 |
377 |    OUTPUT = 'SOME STATISTICS ON THE GRAMMAR:';   DOUBLE_SPACE;
378 |    OUTPUT = '   NUMBER OF TERMINAL SYMBOLS = ' || #_TERMINALS;
379 |    OUTPUT = '   NUMBER OF NONTERMINAL SYMBOLS = ' || #_NTS;
380 |    OUTPUT = '         TOTAL NUMBER OF SYMBOLS = ' || #_TERMINALS + #_NTS;
381 |    OUTPUT = '';
382 |    OUTPUT = '   NUMBER OF PRODUCTIONS = ' || #_PRODS - 1;
383 |       /*  THE PRODUCTION ADDED AUTOMATICALLY IS NOT COUNTED.  */
384 |    N = SHR(#_PRODS, 2);    /*  REALLY (2 * #_PRODS) / 8 FOR
385 |                                          ON_LEFT AND ON_RIGHT.  */
386 |    N = N + PROD_START(#_PRODS) + 3;  /*  FOR PRODUCTIONS, PER SE.  */
387 |    N = N + 5 * #_PRODS;    /*  FOR PROD_START (2 BYTES EACH,
388 |                                 RT_PT_SIZE, AND FIRST_PROD_FOR.  */
389 |    N = N + 4;     /*  FOR #_PRODS, #_TERMINALS, #_NTS, AND GOAL_SYMBOL.  */
390 |    OUTPUT = '   SPACE REQUIRED TO STORE THE PRODUCTIONS = ' || N ||
391 |                        ' BYTES, NOT INCLUDING THE VOCABULARY.';
392 |    LINE = 100* (PROD_START(#_PRODS) + 3 - #_PRODS) / #_PRODS;
393 |    OUTPUT = '   THE AVERAGE LENGTH OF  THE RIGHT PARTS OF PRODUCTIONS = '
394 |          || SUBSTR(LINE, 0, 1) || '.' || SUBSTR(LINE, 1, 2) || ' SYMBOLS.';
395 |    EJECT_PAGE;
396 |  END PRINT_G;    /*************************************************************/
397 |
398 |
399 |
400 | /*THEN A PROCEDURE TO READ A GRAMMAR.  ****************************************/
401 |
402 |READ_G:
403 |   PROCEDURE;
404 |      DECLARE #_SYMS BIT(8),    /*  GLOBAL TO SORT_V, FIND_GOAL, SCAN,        */
405 |               INC_#_SYMS LITERALLY                    /*  AND GETTING_CARDS.  */
406 |               'IF #_SYMS < 255 THEN #_SYMS = #_SYMS + 1;
407 |                ELSE CALL ERROR(''TOO MANY SYMBOLS.'')';
408 |
409 |   /* FIRST SOME USEFUL PROCEDURES.  */
410 |
411 |
412 |   FIND_GOAL:        /*  FIND GOAL SYMBOL, ADD BRACKETTING PRODUCTION.  ***/
413 |      PROCEDURE;
414 |         DECLARE SYM FIXED;
415 |         GOAL_SYMBOL = 0;
416 |         DO SYM = 1 TO #_SYMS;    /*  FIND GOAL SYMBOL.  */
417 |          IF ¬ON_RIGHT(SYM) THEN
418 |             IF GOAL_SYMBOL = 0
419 |             THEN GOAL_SYMBOL = SYM;
420 |             ELSE CALL ERROR('MORE THAN ONE GOAL SYMBOL WAS FOUND:  ' ||
```

```
421 |                         V(GOAL_SYMBOL) || ' USED, ' || V(SYM) || ' IGNORED.';
422 |         END;
423 |         IF GOAL_SYMBOL = 0 THEN                                          DO;
424 |            GOAL_SYMBOL = PROD_ARRAY(PROD_START(1));
425 |            OUTPUT = 'NO EXPLICIT GOAL SYMBOL WAS FOUND.  ' ||
426 |                     V(GOAL_SYMBOL) || ' WILL BE USED AS THE GOAL SYMBOL.';
427 |            DOUBLE_SPACE;                                                 END;
428 |
429 |      /* NOW ADD <SYSTEM_GS> ::=  _|_ <GOAL_SYMBOL> _|_  TO THE GRAMMAR.  */
430 |         INC_#_PRODS;
431 |         PROD_START(#_PRODS) = PROD_ARRAY_PTR;
432 |         RT_PT_SIZE(#_PRODS) = 3;
433 |         PROD_ARRAY(PROD_ARRAY_PTR) = 0;     /*  <SYSTEM_GS>  */
434 |         INC_PROD_ARRAY_PTR;
435 |         PROD_ARRAY(PROD_ARRAY_PTR) = 1;     /*  V(1) = '_|_'  */
436 |         INC_PROD_ARRAY_PTR;
437 |         PROD_ARRAY(PROD_ARRAY_PTR) = GOAL_SYMBOL;
438 |         INC_PROD_ARRAY_PTR;
439 |         PROD_ARRAY(PROD_ARRAY_PTR) = 1;     /*  V(1) = '_|_'  */
440 |      END FIND_GOAL;     /*************************************************/
441 |
442 |
443 |   SORT_V:               /*  SORT THE VOCABULARY: TERMINALS FIRST.  **********/
444 |                         /*  ALSO SET #_TERMINALS AND #_NTS, AND    */
445 |      PROCEDURE;                      /*  FILL IN FIRST_PROD_FOR.  */
446 |
447 |         DECLARE INDEX(255) BIT(8), NEW_INDEX(255) BIT(8),
448 |                 (I, J, TEMP) FIXED, T CHARACTER;
449 |
450 |      /* RECORD THE INDICES.  */
451 |         DO I = 1 TO #_SYMS;     INDEX(I) = I;     END;
452 |
453 |      /* BUBBLE SORT.  */
454 |         DO I = 3 TO #_SYMS;              /*  V(0) AND V(1) ARE SYSTEM SYMBOLS.  */
455 |            J = #_SYMS;
456 |            DO WHILE J >= I;
457 |             IF ON_LEFT(J - 1) & ¬ON_LEFT(J) THEN                         DO;
458 |                ON_LEFT(J - 1) = FALSE;   ON_LEFT(J) = TRUE;
459 |                /*  ON_RIGHT BECOMES MEANINGLESS.  */
460 |                TEMP = INDEX(J - 1); INDEX(J - 1) = INDEX(J); INDEX(J) = TEMP;
461 |                T = V(J - 1);   V(J - 1) = V(J);   V(J) = T;              END;
462 |             J = J - 1;
463 |            END;
464 |         END;
465 |
466 |      /* COMPUTE NEW INDICES.  */
467 |         DO I = 1 TO #_SYMS;
468 |            NEW_INDEX(INDEX(I)) = I;
469 |         END;
470 |
471 |      /* SUBSTITUTE THE NEW INDICES INTO THE PRODUCTIONS.  */
472 |         GOAL_SYMBOL = NEW_INDEX(GOAL_SYMBOL);
473 |         DO I = 1 TO PROD_START(#_PRODS) + 3;
474 |            PROD_ARRAY(I) = NEW_INDEX(PROD_ARRAY(I));
475 |         END;
476 |
477 |      /* ALSO FILL IN FIRST_PROD_FOR.  */
478 |         DO I = 1 TO #_PRODS;
479 |            J = PROD_START(I);
480 |            IF PROD_ARRAY(J) ¬= PROD_ARRAY(PROD_START(I - 1)) THEN
481 |               FIRST_PROD_FOR(PROD_ARRAY(J)) = I;
```

```
482 |       END;
483 |
484 |    /* COUNT THE SYMBOLS.  */
485 |       #_TERMINALS = 1;
486 |       DO WHILE ¬ON_LEFT(#_TERMINALS + 1);
487 |          #_TERMINALS = #_TERMINALS + 1;
488 |       END;
489 |       IF #_TERMINALS > 127 THEN CALL ERROR('TOO MANY TERMINAL SYMBOLS.');
490 |       #_NTS = #_SYMS - #_TERMINALS;
491 |       IF #_NTS > 127 THEN CALL ERROR('TOO MANY NONTERMINAL SYMBOLS.');
492 |
493 |    END SORT_V;    /*********************************************************/
494 |
495 |
496 |    DECLARE CARD_IMAGE CHARACTER, CARD_PTR FIXED;
497 | SCAN:             /*  GET A SYMBOL FROM THE INPUT CARD IMAGE.  *********/
498 |    PROCEDURE BIT(8);
499 |
500 |    LOOK_UP:          /*  GET INDEX OF SYMBOL IN V.  ****************/
501 |       PROCEDURE (SYMBOL) BIT(8);
502 |          DECLARE SYMBOL CHARACTER;
503 |
504 |          DECLARE J FIXED;
505 |          DO J = 2 TO #_SYMS;    /*  IS SYMBOL ALREADY IN V?  */
506 |             IF V(J) = SYMBOL THEN RETURN J;
507 |          END;    /*  V(0) AND V(1) ARE SYSTEM SYMBOLS.  */
508 |
509 |          INC_#_SYMS;          /*  NO, SO ADD SYMBOL TO V.  */
510 |          V(#_SYMS) = SYMBOL;
511 |          RETURN #_SYMS;
512 |       END LOOK_UP;    /*************************************************/
513 |
514 |
515 |    /* THE BODY OF THE PROCEDURE "SCAN".  ****************************/
516 |
517 |       DECLARE LEFT_BRACKET LITERALLY 'BYTE(''<'')',
518 |               RIGHT_BRACKET LITERALLY 'BYTE(''>'')',
519 |               END_OF_CARD LITERALLY '79';
520 |
521 |       DO CARD_PTR = CARD_PTR TO END_OF_CARD;
522 |         /* WATCH FOR ABNORMAL EXITS FROM THIS LOOP.  */
523 |         IF BYTE(CARD_IMAGE, CARD_PTR) ¬= BLANK THEN                    DO;
524 |
525 |            DECLARE (LP, STOP) FIXED;
526 |            LP = CARD_PTR;    /*  MARK LEFT BOUNDARY.  */
527 |            IF BYTE(CARD_IMAGE, LP) = LEFT_BRACKET &
528 |               BYTE(CARD_IMAGE, LP + 1) ¬= BLANK
529 |            THEN STOP = RIGHT_BRACKET;
530 |            ELSE STOP = BLANK;
531 |
532 |         /* NOW LOOK FOR STOP.  */
533 |            DECLARE NO_STOP_FOUND BOOLEAN;
534 |            NO_STOP_FOUND = TRUE;
535 |            DO WHILE NO_STOP_FOUND & CARD_PTR < END_OF_CARD;
536 |               CARD_PTR = CARD_PTR + 1;
537 |               IF BYTE(CARD_IMAGE, CARD_PTR) = STOP THEN
538 |                  NO_STOP_FOUND = FALSE;
539 |            END;
540 |
541 |         /* IN CASE MATCHING RIGHT BRACKET IS NOT FOUND.  */
542 |            IF NO_STOP_FOUND & STOP = RIGHT_BRACKET THEN                DO;
```

```
543 |                  CALL ERROR('UNMATCHED BRACKET: <');
544 |                  CARD_PTR = LP;              /*  ERROR RECOVERY.  */
545 |                  CARD_IMAGE = CARD_IMAGE || ' ';
546 |                  DO WHILE BYTE(CARD_IMAGE, CARD_PTR) ¬= BLANK;
547 |                     CARD_PTR = CARD_PTR + 1;
548 |                  END;                                                    END;
549 |
550 |            /* GET THE SYMBOL AND RETURN.  */
551 |               DECLARE SYMBOL CHARACTER;
552 |               IF STOP = RIGHT_BRACKET | CARD_PTR = END_OF_CARD THEN
553 |                  CARD_PTR = CARD_PTR + 1;    /*  PICK UP LAST CHARACTER.  */
554 |               SYMBOL = SUBSTR(CARD_IMAGE, LP, CARD_PTR - LP);
555 |               RETURN LOOK_UP(SYMBOL);                                   END;
556 |            END;
557 |            RETURN 0;     /*  WHEN AT END OF CARD.  */
558 |         END SCAN;     /*************************************************/
559 |
560 |
561 |      GETTING_CARDS:     /*  READ GRAMMAR CARDS.  ************************/
562 |         PROCEDURE BOOLEAN;
563 |
564 |            CARD_PTR = 0;
565 |            DO WHILE TRUE;     /*  ALL EXITS ARE ABNORMAL.  */
566 |               CARD_IMAGE = INPUT;     /*  GET THE CARD.  */
567 |               IF LENGTH(CARD_IMAGE) = 0 THEN                            DO;
568 |                  /*  END OF FILE DETECTED.  */
569 |                  MORE_GRAMMARS = FALSE;
570 |                  IF CONTROL(@I) THEN EJECT_PAGE;
571 |                  RETURN FALSE;                                           END;
572 |               IF CONTROL(@I) THEN OUTPUT = CARD_IMAGE;
573 |               IF BYTE(CARD_IMAGE) = AT
574 |               THEN     /*  CONTROL CARD OR COMMENT.  */                 DO;
575 |                  IF SUBSTR(CARD_IMAGE, 1, 3) = 'EOG' THEN               DO;
576 |                     IF CONTROL(@I) THEN EJECT_PAGE;
577 |                     RETURN FALSE;                                        END;
578 |                  CONTROL(BYTE(CARD_IMAGE, 1)) = ¬CONTROL(BYTE(CARD_IMAGE, 1));END;
579 |               ELSE IF CARD_IMAGE ¬= BLANK_CARD THEN RETURN TRUE;
580 |            END;
581 |         END GETTING_CARDS;     /*************************************************/
582 |
583 |
584 |
585 |      /* NOW THE BODY OF THE PROCEDURE "READ_G".  ************************/
586 |
587 |      /* INITIALIZATION.  */
588 |         DECLARE I FIXED;
589 |         DO I = 0 TO 255;
590 |            ON_LEFT(I), ON_RIGHT(I) = FALSE;
591 |         END;
592 |         V(0) = '<SYSTEM_GS>';   ON_LEFT(0) = TRUE;     /*  A SYSTEM SYMBOL.  */
593 |         V(1) = '_|_';           ON_RIGHT(1) = TRUE;    /*  A SYSTEM SYMBOL.  */
594 |         #_SYMS = 1;       /*  <SYSTEM_GS> IS NOT COUNTED IN #_SYMS.  */
595 |         #_PRODS, PROD_START(0), PROD_ARRAY(0) = 0;
596 |
597 |      /* NOW READ THE CARDS.  */
598 |         PROD_ARRAY_PTR = 1;
599 |         DO WHILE GETTING_CARDS;     /*  WATCH SIDE EFFECTS.  */
600 |
601 |            INC_#_PRODS:     /*  ADD A PRODUCTION.  */
602 |            IF BYTE(CARD_IMAGE, 0) = BLANK
603 |               THEN I = PROD_ARRAY(PROD_START(#_PRODS - 1));
```

```
604 |          ELSE I = SCAN;
605 |       ON_LEFT(I) = TRUE;
606 |       PROD_START(#_PRODS) = PROD_ARRAY_PTR;
607 |       RT_PT_SIZE(#_PRODS) = -1;
608 |
609 |       DO WHILE I > 0;     /*  I = 0 IMPLIES END OF CARD REACHED BY SCAN.  */
610 |          PROD_ARRAY(PROD_ARRAY_PTR) = I;     /*  RECORD THE PRODUCTION.  */
611 |          INC_PROD_ARRAY_PTR;
612 |          RT_PT_SIZE(#_PRODS) = RT_PT_SIZE(#_PRODS) + 1;
613 |          I = SCAN;
614 |          ON_RIGHT(I) = TRUE;
615 |       END;
616 |       IF ERROR_COUNT > 15 THEN                                          DO;
617 |          OUTPUT = 'TOO MANY ERRORS.  EXECUTION TERMINATED FOR THIS GRAMMAR.';
618 |          DO WHILE GETTING_CARDS;   END;                                 END;
619 |    END;
620 |    CALL FIND_GOAL;
621 |    CALL SORT_V;
622 |  END READ_G;    /*************************************************************/
623 |
624 |
625 |
626 | /*THE REPRESENTATION OF A CHARACTERISTIC FSM.  ********************************/
627 |
628 |  DECLARE READ_XITIONS_TABLE(32767)   BIT(8),
629 |          SYM_BEFORE         (255)    BIT(8),
630 |          XLATION_RULE       (127)    BIT(8),
631 |          #_TO_POP           (127)    BIT(8),
632 |          SYM_TO_READ        (127)    BIT(8),
633 |          DEFAULT_XITION     (127)    BIT(8),
634 |          IS_INADEQUATE      (127)    BOOLEAN,
635 |          IS_REDUCE          (127)    BOOLEAN,
636 |          (IS_LR0, IS_SLR1)           BOOLEAN,
637 |          EXIT_STATE         LITERALLY '0',    /*  A REDUCE STATE.  */
638 |          ERROR_STATE        LITERALLY '0',    /*  A READ STATE.  */
639 |          IS_LA_REDUCE       LITERALLY '128 >',
640 |          IS_READ            LITERALLY '127 <',    /*  ALSO,
641 |          V                  (255)    CHARACTER,      AND
642 |          (#_TERMINALS,#_NTS,#_PRODS) FIXED,          ARE PART OF BOTH THE
643 |                                                      GRAMMAR AND THE CFSM.  */
644 |          (#_READ_STATES, #_LA_REDUCE_STATES) FIXED,
645 |
646 |          INC_#_LA_REDUCE_STATES LITERALLY
647 |          'IF #_LA_REDUCE_STATES = 127
648 |           THEN CALL ERROR(''TOO MANY LOOK-AHEAD--REDUCE STATES.'');
649 |           ELSE #_LA_REDUCE_STATES = #_LA_REDUCE_STATES + 1',
650 |
651 |          INC_#_READ_STATES LITERALLY
652 |          'IF #_READ_STATES = 127
653 |           THEN CALL ERROR(''TOO MANY READ STATES.'');
654 |           ELSE #_READ_STATES = #_READ_STATES + 1';
655 |
656 |  READ_XITION:    /*  ACCESS A READ TRANSITION.  */
657 |     PROCEDURE (STATE, SYM) BIT(8);
658 |        DECLARE (STATE, SYM) BIT(8);
659 |        RETURN READ_XITIONS_TABLE(SHL(STATE, 8) | SYM);
660 |     END READ_XITION;
661 |
662 |  UPDATE_READ_XITION:
663 |     PROCEDURE (STATE1, SYM, STATE2);
664 |        DECLARE (STATE1, SYM, STATE2) BIT(8);
```

```
665 |          READ_XITIONS_TABLE(SHL(STATE1, 8) | SYM) = STATE2;   RETURN;
666 |       END UPDATE_READ_XITION;
667 |
668 |       /*
669 |    FLAG:  */     /*  READ STATES ARE FLAGGED IN THE TABLES.  */
670 |       DECLARE FLAG LITERALLY '"(1)10000000" |';
671 |
672 |       /*
673 |    UNFLAG:  */
674 |       DECLARE UNFLAG LITERALLY '"(1)01111111" &';
675 |
676 |    I_TO_S4:
677 |       PROCEDURE (STATE) CHARACTER;
678 |          DECLARE STATE FIXED;
679 |          DECLARE S CHARACTER;
680 |          IF IS_LA_REDUCE(STATE)
681 |          THEN                                                          DO;
682 |             IF STATE = EXIT_STATE THEN RETURN 'EXIT';
683 |             S = STATE;     /*  CONVERT STATE TO CHARACTER.  */
684 |             RETURN SUBSTR(X12, 0, 4 - LENGTH(S)) || S;                 END
685 |          ELSE                                                          DO;
686 |             STATE = UNFLAG(STATE);
687 |             IF STATE = ERROR_STATE THEN RETURN '    ';
688 |             S = '*' || STATE;     /*  THE * INDICATES A READ STATE.  */
689 |             IF LENGTH(S) = 4 THEN RETURN S;
690 |             RETURN SUBSTR(X12, 0, 4 - LENGTH(S)) || S;                 END;
691 |       END I_TO_S4;
692 |
693 |
694 |
695 |PRINT_CFSM:     /*  A PROCEDURE TO PRINT THE CHARACTERISTIC FSM.  **************/
696 |    PROCEDURE;
697 |
698 |    STR_LNTH_8:
699 |       PROCEDURE (STRING) CHARACTER;
700 |          DECLARE STRING CHARACTER;
701 |          IF LENGTH(STRING) < 4
702 |          THEN RETURN SUBSTR(X12, 0, 4 - LENGTH(STRING)) || STRING || '    ';
703 |          ELSE IF LENGTH(STRING)  >= 7
704 |               THEN RETURN SUBSTR(STRING, 0, 7) || ' ';
705 |               ELSE RETURN STRING || SUBSTR(X12, 0, 8 - LENGTH(STRING));
706 |       END STR_LNTH_8;
707 |
708 |
709 |    /* THE BODY OF THE PROCEDURE "PRINT_CFSM".  ****************************/
710 |
711 |       DECLARE (I, J) FIXED, (LINE1, LINE2) CHARACTER;
712 |       DECLARE VAR CHARACTER;
713 |       OUTPUT = 'THE CFSM FOR THE GRAMMAR IS AS FOLLOWS:';
714 |       DOUBLE_SPACE;
715 |
716 |       /* FIRST PRINT THE LOOK-AHEAD--REDUCE TRANSITIONS.  */
717 |       OUTPUT = X12 ||
718 |       '    T H E   L O O K - A H E A D -- R E D U C E   T R A N S I T I O N S';
719 |       DOUBLE_SPACE;
720 |       OUTPUT = 'LOCK-AHEAD   |  DEFAULT       NUMBER    SYMBOL';
721 |       OUTPUT = '  REDUCE     | TRANSITION   OF STATES     TO  ';
722 |       OUTPUT = '  STATE      |  TO STATE      TO POP     READ ';
723 |       DOUBLE_SPACE;
724 |       DECLARE STATE FIXED;
725 |       DO STATE = 1 TO #_LA_REDUCE_STATES;
```

```
726 |          VAR = I_TO_S4(DEFAULT_XITION(STATE));
727 |          LINE1 = '     |      ' || VAR || '           ';
728 |          LINE1 = LINE1 || #_TO_POP(STATE) || '        ' || V(SYM_TO_READ(STATE));
729 |          CALL LINE_OUT(STATE, LINE1);
730 |       END;
731 |       IF #_LA_REDUCE_STATES > 21 THEN EJECT_PAGE;   ELSE DOUBLE_SPACE;
732 |
733 |       /* NEXT PRINT THE READ TRANSITIONS.  */
734 |       OUTPUT = '                          T H E   R E A D   T R A N S I T I O N S';
735 |       DOUBLE_SPACE;
736 |       OUTPUT = '           SYMBOLS';   OUTPUT = '';
737 |
738 |       DO I = 30 TO #_TERMINALS + #_NTS + 29 BY 30;
739 |          LINE1 = 'READ       ';   LINE2 = 'STATES         ';
740 |          DO J = I - 29 TO MIN(I, #_TERMINALS + #_NTS);
741 |             VAR = STR_LNTH_8(V(J));
742 |             IF ODD(J) THEN LINE1 = LINE1 || VAR;
743 |                       ELSE LINE2 = LINE2 || VAR;
744 |          END;
745 |          OUTPUT = LINE1 ; OUTPUT = LINE2;
746 |          OUTPUT = SUBSTR(DASHED_LINE, 0,
747 |                                10 + 4*MIN(30, #_TERMINALS + #_NTS + 30 - I));
748 |
749 |          DO STATE = 1 TO #_READ_STATES;
750 |             VAR = I_TO_S4(FLAG(STATE));
751 |             LINE1 = '  ' || VAR || '   |';
752 |             DO J = I - 29 TO MIN(I, #_TERMINALS + #_NTS);
753 |                VAR = I_TO_S4(READ_XITION(STATE, J));
754 |                LINE1 = LINE1 || VAR;
755 |             END;
756 |             OUTPUT = LINE1;
757 |          END;
758 |          IF #_LA_REDUCE_STATES + #_READ_STATES > 16 THEN EJECT_PAGE;
759 |                                                     ELSE DOUBLE_SPACE;
760 |       END;
761 |
762 |       /* NOW PRINT THE SYMBOLS PRECEDING THE STATES.  */
763 |       OUTPUT = '    T H E   S Y M B O L S   B E F O R E   T H E   S T A T E S';
764 |       DOUBLE_SPACE;
765 |       OUTPUT = ' LA-REDUCE                                  READ';
766 |       OUTPUT = ' STATE   SYMBOL BEFORE THE STATE    STATE   SYMBOL BEFORE' ||
767 |                                                                ' THE STATE';
768 |       DOUBLE_SPACE;
769 |       V(0) = 'ERROR_TOKEN';
770 |       DO STATE = 1 TO MIN(#_LA_REDUCE_STATES, #_READ_STATES);
771 |          LINE1 = I_TO_S4(FLAG(STATE));
772 |          LINE1 = '     '|| LINE1 || '    ' || V(SYM_BEFORE(FLAG(STATE)));
773 |          CALL LINE_OUT(STATE, SUBSTR(V(SYM_BEFORE(STATE)) || HALF_LINE, 0, 23)
774 |                                                                || LINE1);
775 |       END;
776 |       IF #_LA_REDUCE_STATES > #_READ_STATES
777 |       THEN
778 |          DO STATE = STATE TO #_LA_REDUCE_STATES;
779 |             CALL LINE_OUT(STATE, V(SYM_BEFORE(STATE)));
780 |          END;
781 |       ELSE
782 |          DO STATE = STATE TO #_READ_STATES;
783 |             LINE1 = I_TO_S4(FLAG(STATE));
784 |             OUTPUT = SUBSTR(HALF_LINE, 0, 37) || LINE1 || '    ' ||
785 |                                                 V(SYM_BEFORE(FLAG(STATE)));
786 |          END;
```

```
787 |       EJECT_PAGE;
788 |
789 |       /* FINALLY, PRINT SOME STATISTICS ON THE CFSM.  */
790 |       OUTPUT = 'SOME STATISTICS ON THE CFSM:';
791 |       DOUBLE_SPACE;
792 |       OUTPUT = '   NUMBER OF LOOK-AHEAD--REDUCE STATES = '|| #_LA_REDUCE_STATES;
793 |       OUTPUT = '   NUMBER OF READ STATES = ' || #_READ_STATES;
794 |       OUTPUT = '      TOTAL NUMBER OF STATES = ' || #_LA_REDUCE_STATES +
795 |                                                    #_READ_STATES;
796 |       OUTPUT = '';
797 |       OUTPUT = '   SPACE REQUIRED FOR LOOK-AHEAD--REDUCE TRANSITIONS = ' ||
798 |                                       #_LA_REDUCE_STATES * 4 || ' BYTES.';
799 |       I = (#_TERMINALS + 7) / 8 * #_NTS;
800 |       OUTPUT = '   SPACE REQUIRED FOR LOOK-AHEAD SETS = ' || I || ' BYTES.';
801 |       J = #_READ_STATES * (#_TERMINALS + #_NTS + 1);
802 |       OUTPUT = '   SPACE REQUIRED FOR READ TRANSITIONS = ' || J ||.' BYTES.';
803 |       OUTPUT = '      TOTAL SPACE REQUIRED FOR THE CFSM = ' || I + J + 4 *
804 |                #_LA_REDUCE_STATES || ' BYTES, NOT INCLUDING THE VOCABULARY.';
805 |       DOUBLE_SPACE;   DOUBLE_SPACE;
806 |    END PRINT_CFSM;    /*************************************************************/
807 |
808 |
809 |
810 | /*A PROCEDURE TO COMPUTE THE GRAMMAR'S CHARACTERISTIC FSM.  ******************/
811 |
812 |COMPUTE_CFSM:
813 |    PROCEDURE;
814 |
815 |       /*  FIRST SOME USEFUL DEFINITIONS.  ***********************************/
816 |
817 |    MAKE_CONFIG:
818 |       PROCEDURE (P, DOT) BIT(16);
819 |          DECLARE (P, DOT) BIT(8);
820 |          RETURN SHL(P, 8) | DOT;
821 |       END MAKE_CONFIG;
822 |
823 |
824 |    PROD:
825 |       PROCEDURE (CONFIG) BIT(8);
826 |          DECLARE CONFIG BIT(16);
827 |          RETURN SHR(CONFIG, 8);
828 |       END PROD;
829 |
830 |    DOT:
831 |       PROCEDURE (CONFIG) BIT(8);
832 |          DECLARE CONFIG BIT(16);
833 |          RETURN CONFIG & "00FF";
834 |       END DOT;
835 |
836 |
837 |    SYM_AFTER_DOT:
838 |       PROCEDURE (CONFIG) BIT(8);
839 |          DECLARE CONFIG BIT(16);
840 |          IF DOT(CONFIG) < RT_PT_SIZE(PROD(CONFIG))
841 |          THEN RETURN PROD_ARRAY(PROD_START(PROD(CONFIG)) + DOT(CONFIG) + 1);
842 |          ELSE RETURN 0;    /*  INDICATES DOT IS AT END OF THE PRODUCTION.  */
843 |       END SYM_AFTER_DOT;
844 |
845 |       /*
846 |    DOT_AT_END:  */
847 |       DECLARE DOT_AT_END LITERALLY '0 = SYM_AFTER_DOT';
```

```
848 |
849 |       /*
850 |    SUCCESSOR:  */
851 |       DECLARE SUCCESSOR LITERALLY '1+';
852 |
853 |
854 |
855 |    /* SPACE USED IN COMPUTING THE CFSM.  ************************************/
856 |
857 |       DECLARE BASIS_STACK  (511) BIT(16),
858 |               BS_START     (255) BIT(16),
859 |               BS_SIZE      (255) BIT(8),
860 |               CONFIG_SET   (127) BIT(16),
861 |               #_CONFIGS          BIT(8);
862 |
863 |       DECLARE INC_#_CONFIGS LITERALLY
864 |       'IF #_CONFIGS < 127 THEN #_CONFIGS = #_CONFIGS + 1;   ELSE DO;
865 |        CALL ERROR(''THE CONFIGURATION SET FOR STATE '' || STATE ||
866 |        '' IS TOO LARGE.''):   CONTROL(aC) = TRUE;   END';
867 |
868 |
869 |    COMPUTE_CONFIG_SET:        /*  A PROCEDURE TO COMPUTE   ****************/
870 |       PROCEDURE (STATE);                          /*  CONFIGURATION SETS.  */
871 |
872 |          DECLARE STATE FIXED;    /*  GLOBAL TO PRINT_CS AND ADD_SUCCS_TO_CS.  */
873 |
874 |
875 |       PRINT_CS:              /*  A PROCEDURE TO PRINT   ****************/
876 |          PROCEDURE;                            /*  CONFIGURATION SETS.  */
877 |
878 |             DECLARE (I,J,P) FIXED, LINE CHARACTER;
879 |             OUTPUT = 'THE CONFIGURATION SET FOR STATE '||STATE||' IS:';
880 |             DO I = 1 TO #_CONFIGS;
881 |                P = PROD (CONFIG_SET(I));
882 |                LINE = V(LEFT_PART (P))) || '  ->  ';
883 |                DO J = 1 TO RT_PT_SIZE (P);
884 |                   IF DOT (CONFIG_SET (I)) = J-1 THEN LINE = LINE ||' .';
885 |                   LINE = LINE ||' '|| V (PROD_ARRAY (PROD_START (P)+J));
886 |                END;
887 |                IF DOT(CONFIG_SET(I)) = J - 1 THEN LINE = LINE || ' .';
888 |                CALL LINE_OUT (I, LINE);
889 |             END;
890 |             DOUBLE_SPACE;
891 |          END PRINT_CS;    /*********************************************/
892 |
893 |
894 |       ADD_SUCCS_TO_CS:    /*  A PROCEDURE TO COMPUTE CLOSURE SETS.  ***/
895 |          PROCEDURE(SYM);
896 |             DECLARE SYM FIXED;
897 |             DECLARE I FIXED:   I = 0;
898 |             DO WHILE LEFT_PART(FIRST_PROD_FOR(SYM) + I)) = SYM;
899 |                INC_#_CONFIGS;
900 |                CONFIG_SET(#_CONFIGS) = MAKE_CONFIG(FIRST_PROD_FOR(SYM) + I, 0);
901 |                I = I + 1;
902 |             END;
903 |          END ADD_SUCCS_TO_CS:    /*************************************/
904 |
905 |
906 |       SORT_CS:                /*  SORT A CONFIGURATION SET.  *********/
907 |          PROCEDURE;
908 |             DECLARE (I, J, K, SYM, TEMP) FIXED;
```

```
909 |
910 |          /* MOVE LATER INSTANCES OF A SYMBOL UP WITH IT'S FIRST INSTANCE.  */
911 |             DO I = 0 TO #_CONFIGS - 1;     /*  BUBBLE SORT.  */
912 |                 IF I = 0 THEN SYM = 0;/*CONFIGURATIONS WITH DOT AT END AT TOP.*/
913 |                           ELSE SYM = SYM_AFTER_DOT(CONFIG_SET(I));
914 |                 IF SYM ¬= SYM_AFTER_DOT(CONFIG_SET(I + 1)) THEN
915 |                   DO J = I + 2 TO #_CONFIGS;
916 |                     IF SYM = SYM_AFTER_DOT(CONFIG_SET(J)) THEN                DO;
917 |                        TEMP = CONFIG_SET(J);
918 |                        K = J;
919 |                        DO WHILE K > I + 1;
920 |                           CONFIG_SET(K) = CONFIG_SET(K-1);
921 |                           K = K - 1;
922 |                        END;
923 |                        CONFIG_SET(K) = TEMP;     /*  K = I + 1.  */
924 |                        I = K;     /*  INCREMENT I.  */                      END;
925 |                   END;
926 |             END;
927 |          END SORT_CS;     /*********************************************/
928 |
929 |
930 |       /* THE BODY OF THE PROCEDURE "COMPUTE_CONFIG_SET".  *************/
931 |
932 |          /*  FIRST PICK UP THE BASIS SET.  */
933 |          DO #_CONFIGS = 0 TO BS_SIZE(STATE) - 1;
934 |             CONFIG_SET(#_CONFIGS+1) = BASIS_STACK(BS_START(STATE) + #_CONFIGS);
935 |          END;
936 |
937 |          /*  NOW COMPUTE THE CLOSURE SET;  ADD IT TO THE CONFIGURATION SET.  */
938 |          DECLARE NOT_DONE(255) BOOLEAN, SYM FIXED;
939 |          DO SYM = 0 TO #_TERMINALS + #_NTS;
940 |             NOT_DONE(SYM) = TRUE;     /*  NO SUCCESSORS YET.  */
941 |          END;
942 |
943 |          DECLARE I FIXED;    I = 1;
944 |          DO WHILE I ¬> #_CONFIGS;
945 |             SYM = SYM_AFTER_DOT(CONFIG_SET(I));
946 |             IF IS_NONTERMINAL(SYM) & NOT_DONE(SYM) & SYM ¬= 0 THEN          DO;
947 |                CALL ADD_SUCCS_TO_CS(SYM);     /*  INCREMENTS #_CONFIGS.  */
948 |                NOT_DONE(SYM) = FALSE;     /*  THIS SYMBOL HAS BEEN DONE.  */ END;
949 |             I = I + 1;
950 |          END;
951 |          IF #_CONFIGS > 1 THEN CALL SORT_CS;
952 |          IF CONTROL(2C) THEN CALL PRINT_CS;
953 |       END COMPUTE_CONFIG_SET;     /*********************************************/
954 |
955 |
956 |
957 |       DECLARE READ_TO_INAD(127) BIT(8);
958 |       DECLARE STATE_PTRS  (127) BIT(8);
959 |    ADD_SUCCESSORS_TO_BS:      /*  A PROCEDURE TO COMPUTE THE BASIS SETS   **/
960 |       PROCEDURE (STATE);        /*  OF STATES THAT ARE SUCCESSORS OF STATE.*/
961 |          DECLARE STATE FIXED;
962 |
963 |          DECLARE BASIS_SET(15) BIT(16);
964 |          DECLARE INC_BASIS_SET_SIZE LITERALLY
965 |          'IF BASIS_SET_SIZE < 15 THEN BASIS_SET_SIZE = BASIS_SET_SIZE + 1;  ELSE
966 |           CALL ERROR(''THE BASIS SET FOR STATE '' ||STATE|| '' IS TOO BIG.'')';
967 |
968 |       LOOK_UP_READ:      /*  GET INDEX OF BASIS_SET IN BS_STACK.  ******/
969 |          PROCEDURE (SET_SIZE) BIT(8);
```

```
970 |            DECLARE SET_SIZE FIXED;                                                    |
971 |                                                                                        |
972 |            DECLARE (I, J, STATE) FIXED, (ALL_SAME, ONE_SAME) BOOLEAN;                 |
973 |            DO STATE = 1 TO #_READ_STATES;                                              |
974 |              IF BS_SIZE(STATE) = SET_SIZE THEN                                   DO;   |
975 |                 ALL_SAME = TRUE;                                                       |
976 |                 DO I = BS_START(STATE) TO BS_START(STATE) + SET_SIZE - 1;               |
977 |                    ONE_SAME = FALSE;                                                   |
978 |                    DO J = 1 TO SET_SIZE;                                               |
979 |                       ONE_SAME = ONE_SAME|BASIS_STACK(I) = BASIS_SET(J);               |
980 |                    END;                                                                |
981 |                    ALL_SAME = ALL_SAME & ONE_SAME;                                     |
982 |                 END;                                                                   |
983 |              IF ALL_SAME THEN RETURN FLAG(STATE);                                END;  |
984 |            END;                                                                        |
985 |                                                                                        |
986 |         /* GET HERE ONLY IF BASIS_SET IS NOT FOUND IN BS_STACK.  */                    |
987 |            INC_#_READ_STATES;    /*  ADD BASIS_SET TO BS_STACK.  */                    |
988 |            DECLARE VAR FIXED;                                                          |
989 |            VAR = BS_START(STATE - 1);                                                  |
990 |            BS_START(STATE) = VAR + BS_SIZE(STATE - 1);                                 |
991 |            BS_SIZE(STATE) = SET_SIZE;                                                  |
992 |            IF BS_START(STATE) + SET_SIZE > 511 THEN                              DO;   |
993 |                CALL ERROR ('CFSM(THE SET OF BASIS SETS) IS TOO LARGE.');               |
994 |                BS_START(STATE) = 512 - SET_SIZE;                                 END;  |
995 |            DO I = 1 TO SET_SIZE;                                                       |
996 |                BASIS_STACK(BS_START(STATE) + I - 1) = BASIS_SET(I);                     |
997 |            END;                                                                        |
998 |            RETURN FLAG(STATE);                                                         |
999 |         END LOOK_UP_READ;     /*****************************************/              |
1000|                                                                                        |
1001|                                                                                        |
1002|      LOOK_UP_REDUCE:    /*  GET INDEX OF A REDUCE STATE.  ************/              |
1003|         PROCEDURE (P) BIT(8);                                                          |
1004|            DECLARE P BIT(8);                                                          |
1005|            IF P = #_PRODS THEN RETURN EXIT_STATE;                                     |
1006|            IF STATE_PTRS(P) = 0 THEN                                            DO;   |
1007|               INC_#_LA_REDUCE_STATES;                                                 |
1008|               STATE_PTRS(P) = #_LA_REDUCE_STATES;                                     |
1009|               IS_REDUCE(#_LA_REDUCE_STATES) = TRUE;                                   |
1010|               DEFAULT_XITION(#_LA_REDUCE_STATES) = FLAG(ERROR_STATE);                 |
1011|               XLATION_RULE(#_LA_REDUCE_STATES) = P;                                   |
1012|               #_TO_POP(#_LA_REDUCE_STATES) = RT_PT_SIZE(P) - 1;                       |
1013|               SYM_TO_READ(#_LA_REDUCE_STATES) = LEFT_PART(P));                  END;  |
1014|            RETURN STATE_PTRS(P);                                                      |
1015|         END LOOK_UP_REDUCE;    /*****************************************/            |
1016|                                                                                        |
1017|                                                                                        |
1018|      /* THE BODY OF THE PROCEDURE "ADD_SUCCESSORS_TO_BS".  ***********/              |
1019|                                                                                        |
1020|         /*  FIRST PROCESS REDUCE TRANSITIONS, IF ANY.  */                             |
1021|         DECLARE I FIXED;   I = 1;                                                      |
1022|         DO WHILE I ¬> #_CONFIGS & DOT_AT_END(CONFIG_SET(I));                          |
1023|            DECLARE P BIT(8);   P = PROD(CONFIG_SET(I));                               |
1024|            INC_#_LA_REDUCE_STATES;                                                    |
1025|            #_TO_POP(#_LA_REDUCE_STATES) = RT_PT_SIZE(P) - 1;                         |
1026|            SYM_TO_READ(#_LA_REDUCE_STATES) = LEFT_PART(P));                          |
1027|            SYM_BEFORE(#_LA_REDUCE_STATES) = SYM_BEFORE(FLAG(STATE));                 |
1028|            IF I = 1          /*  #_CONFIGS WILL BE >1 SINCE SINGLETON                |
1029|                                 CONFIGURATION SETS ARE NEVER GENERATED.  */          |
1030|            THEN                                                                 DO;   |
```

```
1031 |                IS_INADEQUATE(#_LA_REDUCE_STATES) = TRUE;                                    |
1032 |                READ_TO_INAD(STATE) = #_LA_REDUCE_STATES;                          END;      |
1033 |           ELSE IF I < #_CONFIGS                                                             |
1034 |                THEN DEFAULT_XITION(#_LA_REDUCE_STATES-1) = #_LA_REDUCE_STATES;              |
1035 |                ELSE /* THE LAST TRANSITION FROM THIS INADEQUATE  */                        |
1036 |                     /* STATE IS A REDUCE TRANSITION.              */        DO;            |
1037 |                     DECLARE VAR FIXED;   VAR = #_LA_REDUCE_STATES;                         |
1038 |                     DEFAULT_XITION(VAR) = LOOK_UP_REDUCE(P);                   END;        |
1039 |           I = I + 1;                                                                       |
1040 |        END;                                                                                 |
1041 |                                                                                             |
1042 |        /*  FINALLY, PROCESS THE READ TRANSITIONS, IF ANY.  */                              |
1043 |        IF I ¬> #_CONFIGS & I ¬= 1 THEN                                                      |
1044 |           DEFAULT_XITION(#_LA_REDUCE_STATES) = FLAG(STATE);                                |
1045 |        DO WHILE I ¬> #_CONFIGS;                                                             |
1046 |           DECLARE BASIS_SET_SIZE BIT(8);   BASIS_SET_SIZE = 0;                             |
1047 |           DECLARE SYM BIT(8);   SYM = SYM_AFTER_DOT(CONFIG_SET(I));                        |
1048 |           DO WHILE SYM = SYM_AFTER_DOT(CONFIG_SET(I)) & I ¬> #_CONFIGS;                     |
1049 |              INC_BASIS_SET_SIZE;                                                            |
1050 |              BASIS_SET(BASIS_SET_SIZE) = SUCCESSOR(CONFIG_SET(I));                          |
1051 |              I = I + 1;                                                                     |
1052 |           END;                                                                              |
1053 |                                                                                             |
1054 |           DECLARE NEXT_STATE BIT(8);                                                        |
1055 |           IF BASIS_SET_SIZE = 1 & DOT_AT_END(BASIS_SET(1))                                 |
1056 |           THEN NEXT_STATE = LOOK_UP_REDUCE(PROD(BASIS_SET(1)));                            |
1057 |           ELSE NEXT_STATE = LOOK_UP_READ(BASIS_SET_SIZE);                                  |
1058 |           CALL UPDATE_READ_XITION(STATE, SYM, NEXT_STATE);                                 |
1059 |           SYM_BEFORE(NEXT_STATE) = SYM;                                                     |
1060 |        END;                                                                                 |
1061 |     END ADD_SUCCESSORS_TO_BS;     /*****************************************/               |
1062 |                                                                                             |
1063 |                                                                                             |
1064 |                                                                                             |
1065 |  /* THE BODY OF THE PROCEDURE "COMPUTE_CFSM".  ***************************/               |
1066 |                                                                                             |
1067 |     /* INITIALIZATION.  */                                                                |
1068 |     DECLARE I FIXED;                                                                        |
1069 |     DO I = 0 TO 127;                                                                        |
1070 |        READ_TO_INAD(I) = FLAG(I);                                                           |
1071 |        STATE_PTRS(I) = 0;                                                                   |
1072 |        IS_INADEQUATE(I) = FALSE;                                                            |
1073 |        IS_REDUCE(I) = FALSE;                                                                |
1074 |        READ_XITIONS_TABLE(I) = FLAG(ERROR_STATE);                                          |
1075 |     END;                                                                                    |
1076 |     DO I = I TO 32767;                                                                      |
1077 |        READ_XITIONS_TABLE(I) = FLAG(ERROR_STATE);                                          |
1078 |     END;                                                                                    |
1079 |                                                                                             |
1080 |     /*  START WITH <SYSTEM_GS> ::=  . _|_ <GOAL_SYMBOL> _|_  */                           |
1081 |     #_READ_STATES = 1;   #_LA_REDUCE_STATES = 0;                                           |
1082 |     BS_START(#_READ_STATES) = 0;                                                            |
1083 |     BS_SIZE(#_READ_STATES) = 1;                                                             |
1084 |     BASIS_STACK(0) = MAKE_CONFIG(#_PRODS, 0);                                               |
1085 |                                                                                             |
1086 |     /*  ENTER THE EXIT STATE WHEN READY TO REDUCE TO <SYSTEM_GS>.  */                      |
1087 |     STATE_PTRS(#_PRODS) = EXIT_STATE;                                                       |
1088 |                                                                                             |
1089 |     /*  NOW ITERATIVELY COMPUTE THE REST OF THE STATES.  */                                |
1090 |     DECLARE STATE FIXED;   STATE = 1;                                                       |
1091 |     DO WHILE STATE ¬> #_READ_STATES;                                                        |
```

```
1092 |          CALL COMPUTE_CONFIG_SET(STATE);                                    |
1093 |          CALL ADD_SUCCESSORS_TO_BS(STATE);                                  |
1094 |             /*  INCREMENTS #_READ_STATES:  ALSO RECORDS SUCCESSOR  */       |
1095 |             /*  RELATIONS IN READ_XITIONS_TABLE, ETC.              */       |
1096 |          STATE = STATE + 1;                                                 |
1097 |          IF  ERROR_COUNT > 15 THEN                                    DO;   |
1098 |             OUTPUT = 'TOO MANY ERRORS.  EXECUTION TERMINATED FOR THIS CFSM.';|
1099 |             STATE = #_READ_STATES + 1;    /*  TERMINATES DO LOOP.  */   END;|
1100 |       END;                                                                  |
1101 |                                                                             |
1102 |       /*  FIX UP THE TRANSITIONS TO INADEQUATE STATES, IF ANY.  */         |
1103 |       DO STATE = 1 TO #_LA_REDUCE_STATES;                                   |
1104 |          IF IS_INADEQUATE(STATE) THEN                                 DO;   |
1105 |             IS_LRO = FALSE;                                                 |
1106 |             DECLARE(SYM, S) BIT(8);   SYM = SYM_BEFORE(STATE);             |
1107 |             DO S = 1 TO #_READ_STATES;                                      |
1108 |                IF READ_XITION(S, SYM) > 127 THEN                            |
1109 |                CALL UPDATE_READ_XITION(S, SYM, READ_TO_INAD(                |
1110 |                                UNFLAG(READ_XITION(S, SYM))));               |
1111 |             END;                                                      END;  |
1112 |       END;                                                                  |
1113 |    END COMPUTE_CFSM;    /*************************************************/|
1114 |                                                                             |
1115 |                                                                             |
1116 |                                                                             |
1117 | /* THE REPRESENTATION OF SLR(1) LOOK-AHEAD SETS.  ************************/|
1118 |                                                                             |
1119 |    DECLARE LOOK_AHEAD_TABLE(1023) BIT(32);                                  |
1120 |                                                                             |
1121 |       DECLARE MASK LITERALLY '"(1)10000000000000000000000000000000"';       |
1122 |    VALIDATE_LA_XITION:                                                      |
1123 |       PROCEDURE (NT_SYM, SYM);                                              |
1124 |          DECLARE (NT_SYM, SYM) BIT(8), SUB BIT(16), B BIT(32);              |
1125 |          SUB = SHL(NT_SYM-#_TERMINALS, 3) | SHR(SYM, 5);                    |
1126 |          B = LOOK_AHEAD_TABLE(SUB);                                         |
1127 |          B = B | SHR(MASK, SYM & "(1)11111");                               |
1128 |          LOCK_AHEAD_TABLE(SUB) = B;                                         |
1129 |       END;                                                                  |
1130 |                                                                             |
1131 |    IS_VALID_LA:                                                             |
1132 |       PROCEDURE (NT_SYM, SYM) BOOLEAN;                                      |
1133 |          DECLARE (NT_SYM, SYM) BIT(8), SUB BIT(16);                         |
1134 |          SUB = SHL(NT_SYM - #_TERMINALS, 3) | SHR(SYM, 5);                   |
1135 |          RETURN 0 ¬= (LOOK_AHEAD_TABLE(SUB) & SHR(MASK, SYM & "(1)11111"));|
1136 |       END;                                                                  |
1137 |                                                                             |
1138 |    INCLUDED:                                                                |
1139 |       PROCEDURE (SYM, PRED) BOOLEAN;                                        |
1140 |          DECLARE (SYM, PRED) BIT(8),                                        |
1141 |                  (SYM_START, PRED_START, I) BIT(16), INCL BOOLEAN;          |
1142 |          SYM_START = SHL(SYM - #_TERMINALS, 3);                             |
1143 |          PRED_START = SHL(PRED - #_TERMINALS, 3);                           |
1144 |          INCL = TRUE;                                                       |
1145 |          DO I = 0 TO 3;                                                     |
1146 |             DECLARE WORD BIT(32);                                           |
1147 |             WORD = LOOK_AHEAD_TABLE(SYM_START + I) &                        |
1148 |                    LOOK_AHEAD_TABLE(PRED_START + I);                        |
1149 |             INCL = INCL & LOCK_AHEAD_TABLE(PRED_START + I) = WORD;          |
1150 |          END;                                                               |
1151 |          RETURN INCL;                                                       |
1152 |       END INCLUDED;                                                         |
```

```
1153 |
1154 |    INCLUDE:
1155 |       PROCEDURE (SYM, PRED);
1156 |          DECLARE (SYM, PRED) BIT(8),
1157 |                  (SYM_START, PRED_START, I) BIT(16);
1158 |          SYM_START = SHL(SYM - #_TERMINALS, 3);
1159 |          PRED_START = SHL(PRED - #_TERMINALS, 3);
1160 |          DO I = 0 TO 7;
1161 |             DECLARE WORD BIT(32);
1162 |             WORD = LOOK_AHEAD_TABLE(SYM_START + I) |
1163 |                    LOOK_AHEAD_TABLE(PRED_START + I);
1164 |             LOOK_AHEAD_TABLE(SYM_START + I) = WORD;
1165 |          END;
1166 |       END INCLUDE;
1167 |
1168 |    INTERSECT:
1169 |       PROCEDURE (NT_SYM1, NT_SYM2) BOOLEAN;
1170 |          DECLARE (NT_SYM1, NT_SYM2) BIT(8),
1171 |                  (SYM1_START, SYM2_START, I) BIT(16), INTER BOOLEAN;
1172 |          SYM1_START = SHL(NT_SYM1 - #_TERMINALS, 3);
1173 |          SYM2_START = SHL(NT_SYM2 - #_TERMINALS, 3);
1174 |          INTER = FALSE;
1175 |          DO I = 0 TO 3;
1176 |             INTER = INTER | (LOOK_AHEAD_TABLE(SYM1_START + I) &
1177 |                              LOOK_AHEAD_TABLE(SYM2_START + I)  ) ¬=0;
1178 |          END;
1179 |          RETURN INTER;
1180 |       END INTERSECT;
1181 |
1182 |
1183 | /*A PROCEDURE TO PRINT THE SLR(1) LOOK-AHEAD SETS.  ***************************/
1184 |
1185 |PRINT_SLR1_LA_SETS:
1186 |    PROCEDURE;
1187 |
1188 |    STR_NTH13:
1189 |       PROCEDURE (STRING) CHARACTER;
1190 |          DECLARE STRING CHARACTER;
1191 |          IF LENGTH(STRING) >= 11
1192 |          THEN RETURN SUBSTR(STRING, 0, 11) || ' |';
1193 |          ELSE RETURN SUBSTR(X12, 0, 11 - LENGTH(STRING)) || STRING || ' |';
1194 |       END;
1195 |
1196 |       DECLARE LINE CHARACTER, (NT_SYM, T_SYM) BIT(8);
1197 |       IF #_NTS > 18 THEN EJECT_PAGE;
1198 |       OUTPUT = '          T H E   L O O K - A H E A D   S E T S';
1199 |       DOUBLE_SPACE;
1200 |       OUTPUT = 'NONTERMINAL     TERMINAL SYMBOLS';
1201 |       LINE = '  SYMBOLS  ';
1202 |       DO T_SYM = 1 TO #_TERMINALS BY 2;
1203 |          IF T_SYM < 10 THEN LINE = LINE || '   ' || T_SYM;
1204 |                        ELSE LINE = LINE || '  '  || T_SYM;
1205 |       END;
1206 |       OUTPUT = LINE;
1207 |       LINE = '             ';
1208 |       DO T_SYM = 2 TO #_TERMINALS BY 2;
1209 |          IF T_SYM < 10 THEN LINE = LINE || '   ' || T_SYM;
1210 |                        ELSE LINE = LINE || '  '  || T_SYM;
1211 |       END;
1212 |       OUTPUT = LINE;
1213 |       OUTPUT = SUBSTR('   ' || DASHED_LINE, 0, 13 + 2 * #_TERMINALS);
```

```
1214 |       DO NT_SYM = #_TERMINALS + 1 TO #_TERMINALS + #_NTS;
1215 |          LINE = STR_INTH13(V(NT_SYM));
1216 |          DO T_SYM = 1 TO #_TERMINALS;
1217 |             IF IS_VALID_LA(NT_SYM, T_SYM) THEN LINE = LINE || ' 1';
1218 |                                           ELSE LINE = LINE || '  ';
1219 |          END;
1220 |          OUTPUT = LINE;
1221 |       END;
1222 |       EJECT_PAGE;
1223 |    END PRINT_SLR1_LA_SETS;    /*************************************************/
1224 |
1225 |
1226 | /*A PROCEDURE TO COMPUTE SLR(1) LOOK-AHEAD SETS.  ***************************/
1227 |
1228 |COMPUTE_SLR1_LA_SETS:
1229 |    PROCEDURE;
1230 |
1231 |       NOT_SLR1:
1232 |          PROCEDURE (STATE, S);
1233 |             DECLARE (STATE, S) BIT(8);
1234 |             IS_SLR1 = FALSE;
1235 |             OUTPUT = '************************************************************'
1236 |                   || '************************************************************';
1237 |             DOUBLE_SPACE;
1238 |             DECLARE VAR CHARACTER;   VAR = I_TO_S4(S);
1239 |             OUTPUT = 'THE GRAMMAR IS NOT SLR(1):  THE LOOK-AHEAD SET OF STATE '
1240 |                   || STATE || ' INTERSECTS THAT OF STATE ' || VAR || '.';
1241 |             DOUBLE_SPACE;
1242 |             OUTPUT = '************************************************************'
1243 |                   || '************************************************************';
1244 |             DOUBLE_SPACE;
1245 |          END NOT_SLR1;
1246 |
1247 |       DECLARE (STATE, SYM) BIT(8), I FIXED;
1248 |       DO I = 0 TO 1023;     /*  ALL ENTRIES FALSE.  */
1249 |          LOOK_AHEAD_TABLE(I) = "(1)00000000000000000000000000000000";
1250 |       END;
1251 |
1252 |    /* GET THE EXPLICIT LOOK_AHEAD SYMBOLS.  */
1253 |       DO STATE = 2 TO #_READ_STATES;
1254 |          SYM = SYM_BEFORE(FLAG(STATE));
1255 |          IF IS_NONTERMINAL(SYM) THEN                                    DO;
1256 |             DECLARE T_SYM BIT(8);
1257 |             DO T_SYM = 1 TO #_TERMINALS;
1258 |                IF READ_XITION(STATE,T_SYM) ¬= FLAG(ERROR_STATE) THEN
1259 |                CALL VALIDATE_LA_XITION(SYM, T_SYM);
1260 |             END;                                                        END;
1261 |       END;
1262 |
1263 |    /* NOW GET THE SUCCESSOR RELATIONS AMONG NONTERMINALS.  */
1264 |       DO STATE = 1 TO #_LA_REDUCE_STATES;
1265 |          SYM = SYM_BEFORE(STATE);
1266 |          IF IS_NONTERMINAL(SYM) & #_TO_POP(STATE) >= 0 THEN
1267 |          CALL VALIDATE_LA_XITION(SYM, SYM_TO_READ(STATE));
1268 |       END;
1269 |
1270 |    /* COMPLETE THE LOOK-AHEAD SETS.  */
1271 |       DECLARE CHANGING BOOLEAN;   CHANGING = TRUE;
1272 |       DO WHILE CHANGING;
1273 |          CHANGING = FALSE;
1274 |          DECLARE NT_SYM BIT(8);
```

```
1275 |          DO NT_SYM = #_TERMINALS + 1 TO #_TERMINALS + #_NTS;
1276 |             DECLARE PRED_SYM BIT(8);
1277 |             DO PRED_SYM = #_TERMINALS + 1 TO #_TERMINALS + #_NTS;
1278 |                IF IS_VALID_LA(NT_SYM, PRED_SYM) THEN
1279 |                IF ¬INCLUDED(NT_SYM, PRED_SYM) THEN                    DO;
1280 |                   CHANGING = TRUE;
1281 |                   CALL INCLUDE(NT_SYM, PRED_SYM);                     END;
1282 |             END;
1283 |          END;
1284 |       END;
1285 |
1286 |    /* SEE IF THE GRAMMAR IS SLR(1).  */
1287 |       DO STATE = 1 TO #_LA_REDUCE_STATES;
1288 |          IF IS_INADEQUATE(STATE) THEN                                 DO;
1289 |             DECLARE S BIT(8);   S = DEFAULT_XITION(STATE);
1290 |             DO WHILE IS_LA_REDUCE(S) &
1291 |                      ¬INTERSECT(SYM_TO_READ(STATE), SYM_TO_READ(S));
1292 |                S = DEFAULT_XITION(S);
1293 |             END;
1294 |             IF IS_LA_REDUCE(S) THEN CALL NOT_SLR1(STATE, S);
1295 |             ELSE  /*  S IS A READ STATE.  */                         DO;
1296 |                DECLARE DISJOINT BOOLEAN;   DISJOINT = TRUE;
1297 |                DO SYM = 1 TO #_TERMINALS;
1298 |                   IF IS_VALID_LA(SYM_TO_READ(STATE), SYM) THEN
1299 |                   IF READ_XITION(UNFLAG(S), SYM) ¬= (FLAG(ERROR_STATE))
1300 |                   THEN DISJOINT = FALSE;
1301 |                END;
1302 |                IF DISJOINT
1303 |                THEN                                                  DO;
1304 |                   IS_INADEQUATE(STATE) = FALSE;
1305 |                   IF IS_LA_REDUCE(DEFAULT_XITION(STATE)) THEN
1306 |                      IS_INADEQUATE(DEFAULT_XITION(STATE)) = TRUE;     END;
1307 |                ELSE  CALL NOT_SLR1(STATE, S);                      END;END;
1308 |       END;
1309 |    END COMPUTE_SLR1_LA_SETS;    /*************************************************/
1310 |
1311 |
1312 | /*A PROCEDURE TO PUNCH THE DPDA.  ***********************************************/
1313 |
1314 |PUNCH_DPDA:
1315 |    PROCEDURE;
1316 |
1317 |
1318 |       DECLARE CARD_IMAGE CHARACTER;
1319 |       PUNCH_CARD:
1320 |          PROCEDURE (ITEM);
1321 |             DECLARE ITEM CHARACTER;
1322 |             IF LENGTH(CARD_IMAGE) + LENGTH(ITEM) ¬>80
1323 |             THEN CARD_IMAGE = CARD_IMAGE || ITEM;
1324 |             ELSE                                                     DO;
1325 |                 OUTPUT(PUNCH) = CARD_IMAGE;
1326 |                 CARD_IMAGE = '      ' || ITEM;                       END;
1327 |          END PUNCH_CARD;
1328 |
1329 |
1330 |       CONVERT:    /*  CHANGE THE STATE NAMES.  */
1331 |          PROCEDURE (NEXT_STATE) BIT(8);
1332 |             DECLARE NEXT_STATE BIT(8);
1333 |             IF NEXT_STATE = FLAG(ERROR_STATE)
1334 |             THEN NEXT_STATE = #_LA_REDUCE_STATES + #_READ_STATES + 1;
1335 |             ELSE IF NEXT_STATE = EXIT_STATE
```

```
1336 |                THEN NEXT_STATE = #_LA_REDUCE_STATES + #_READ_STATES;                |
1337 |                ELSE IF IS_LA_REDUCE(NEXT_STATE)                                    |
1338 |                     THEN NEXT_STATE = NEXT_STATE - 1;                              |
1339 |                     ELSE NEXT_STATE = #_LA_REDUCE_STATES +                         |
1340 |                                       NEXT_STATE - 129;                            |
1341 |           RETURN NEXT_STATE;                                                       |
1342 |        END CONVERT;                                                                |
1343 |                                                                                    |
1344 |                                                                                    |
1345 |  /* FIRST PUNCH THE VOCABULARY, ETC.  */                                           |
1346 |                                                                                    |
1347 |     DECLARE SYM BIT(8);                                                            |
1348 |                                                                                    |
1349 |     OUTPUT(PUNCH) = '   DECLARE #_TERMINALS LITERALLY ''''||#_TERMINALS||''',':  |
1350 |     OUTPUT(PUNCH) = '           #_NTS       LITERALLY ''''||#_NTS      ||''',':  |
1351 |     OUTPUT(PUNCH) = '           #_SYMS      LITERALLY ''''||                     |
1352 |                                                  #_TERMINALS + #_NTS||''':';      |
1353 |     CARD_IMAGE   =  '   DECLARE V(#_SYMS) CHARACTER INITIAL (';                   |
1354 |     DO SYM = 1 TO #_TERMINALS + #_NTS;                                             |
1355 |        CALL PUNCH_CARD('''' || V(SYM) || ''', ');                                  |
1356 |     END;                                                                           |
1357 |     CALL PUNCH_CARD('''ERROR_TOKEN'');');                                          |
1358 |     CALL PUNCH_CARD(BLANK_CARD);                                                   |
1359 |                                                                                    |
1360 |  /* NEXT PUNCH THE LOOK-AHEAD REDUCE STATES.  */                                   |
1361 |                                                                                    |
1362 |     OUTPUT(PUNCH) = '   DECLARE #_LA_REDUCE_STATES LITERALLY '''                   |
1363 |                                                  || #_LA_REDUCE_STATES || ''';':   |
1364 |     DECLARE VAR FIXED;   VAR = (#_TERMINALS + 7) / 8;                              |
1365 |     OUTPUT(PUNCH) = '   DECLARE #_BYTES_PER_NT LITERALLY ''' || VAR || ''';';     |
1366 |     /* THE SLR(1) LOOK-AHEAD SETS.  */                                            |
1367 |     CARD_IMAGE   =  '   DECLARE LOOK_AHEAD_TABLE('|| #_NTS *                      |
1368 |                          ((#_TERMINALS + 7) / 8) || ') BIT(8) INITIAL (';          |
1369 |     DO SYM = 1 TO #_NTS;                                                           |
1370 |        DECLARE (WORD_#, BITE_#) BIT(8);                                            |
1371 |        DO WORD_# = 0 TO 7;                                                         |
1372 |           DO BITE_# = 0 TO 3;                                                      |
1373 |              IF SHL(WORD_#, 5) + SHL(BITE_#, 3) < #_TERMINALS                      |
1374 |              THEN                                                          DO;     |
1375 |                 DECLARE BITE BIT(8), WORD BIT(32);                                 |
1376 |                 WORD = LOOK_AHEAD_TABLE(WORD_# + SHL(SYM, 3));                     |
1377 |                 IF BITE_# < 3                                                      |
1378 |                 THEN BITE = SHR(WORD, 23 - SHL(BITE_#, 3));                        |
1379 |                 ELSE IF LOOK_AHEAD_TABLE(WORD_# + SHL(SYM, 3) + 1) < 0             |
1380 |                      THEN BITE = SHL(WORD, 1) | 1;                                 |
1381 |                      ELSE BITE = SHL(WORD, 1);                                     |
1382 |                 CALL PUNCH_CARD(BITE || ', ');                             END;    |
1383 |              END;                                                                  |
1384 |           END;                                                                     |
1385 |        END;                                                                        |
1386 |        CALL PUNCH_CARD('0);');                                                     |
1387 |        CALL PUNCH_CARD(BLANK_CARD);                                                |
1388 |                                                                                    |
1389 |        /* THE NUMBER OF SYMBOLS TO POP TO EFFECT A REDUCTION.  */                 |
1390 |        CARD_IMAGE   =  '   DECLARE #_TO_POP(' || #_LA_REDUCE_STATES - 1            |
1391 |                                                 || ') BIT(8) INITIAL (';           |
1392 |        DECLARE STATE BIT(8);                                                       |
1393 |        DO STATE = 1 TO #_LA_REDUCE_STATES - 1;                                     |
1394 |           CALL PUNCH_CARD(#_TO_POP(STATE) || ', ');                                |
1395 |        END;                                                                        |
1396 |        CALL PUNCH_CARD(#_TO_POP(STATE) || ');');                                   |
```

```
1397 |      CALL PUNCH_CARD(BLANK_CARD);
1398 |
1399 |      /* THE SYMBOLS TO READ AFTER REDUCTIONS ARE MADE.  */
1400 |      CARD_IMAGE   =  '   DECLARE SYM_TO_READ(' || #_LA_REDUCE_STATES - 1
1401 |                                                || ') BIT(8) INITIAL (':
1402 |      DO STATE = 1 TO #_LA_REDUCE_STATES - 1;
1403 |         SYM = SYM_TO_READ(STATE);
1404 |         IF SYM = 0 THEN SYM = #_TERMINALS + #_NTS;
1405 |                    ELSE SYM = SYM - 1;
1406 |         CALL PUNCH_CARD(SYM || ', ');
1407 |      END;
1408 |      CALL PUNCH_CARD(SYM_TO_READ(STATE) - 1 || ');');
1409 |      CALL PUNCH_CARD(BLANK_CARD);
1410 |
1411 |      /* THE STATES TO ENTER WHEN LOOK-AHEADS FAIL.  */
1412 |      CARD_IMAGE   =  '   DECLARE DEFAULT_XITION(' || #_LA_REDUCE_STATES - 1
1413 |                                                   || ') BIT(8) INITIAL (';
1414 |      DO STATE = 1 TO #_LA_REDUCE_STATES - 1;
1415 |         CALL PUNCH_CARD(CONVERT(DEFAULT_XITION(STATE)) || ', ');
1416 |      END;
1417 |      CALL PUNCH_CARD(CONVERT(DEFAULT_XITION(STATE)) || ');');
1418 |      CALL PUNCH_CARD(BLANK_CARD);
1419 |
1420 |   /* NOW PUNCH THE READ STATES.  */
1421 |
1422 |      OUTPUT(PUNCH) = '   DECLARE #_READ_STATES LITERALLY '''
1423 |                                                || #_READ_STATES || ''',';
1424 |      OUTPUT(PUNCH) = '           #_LAR_PLUS_READ_STATES LITERALLY '''
1425 |                                 || #_LA_REDUCE_STATES + #_READ_STATES || ''';':
1426 |      CARD_IMAGE   =  '   DECLARE READ_XITIONS_TABLE(' || #_READ_STATES *
1427 |                                 (#_TERMINALS + #_NTS) || ') BIT(8) INITIAL (':
1428 |      DO STATE = 1 TO #_READ_STATES;
1429 |         DO SYM = 1 TO #_TERMINALS + #_NTS;
1430 |            CALL PUNCH_CARD(CONVERT(READ_XITION(STATE, SYM)) || ', ');
1431 |         END;
1432 |      END;
1433 |      CALL PUNCH_CARD('0);');
1434 |      CALL PUNCH_CARD(BLANK_CARD);
1435 |
1436 |   /* PUNCH THE SYMBOLS THAT PRECEDE THE STATES.  */
1437 |
1438 |      CARD_IMAGE   = '   DECLARE SYM_BEFORE(' || #_LA_REDUCE_STATES +
1439 |                                        #_READ_STATES + 1 || ') BIT(8) INITIAL (':
1440 |      DO STATE = 1 TO #_LA_REDUCE_STATES;
1441 |         CALL PUNCH_CARD(SYM_BEFORE(STATE) - 1 || ', ');
1442 |      END;
1443 |      CALL PUNCH_CARD(#_TERMINALS + #_NTS || ', ');    /*  FIRST READ STATE.  */
1444 |      DO STATE = 2 TO #_READ_STATES;
1445 |         CALL PUNCH_CARD(SYM_BEFORE(FLAG(STATE)) - 1 || ', ');
1446 |      END;
1447 |      CALL PUNCH_CARD('0, ' ||
1448 |                      #_TERMINALS + #_NTS || ');'   );
1449 |      CALL PUNCH_CARD(BLANK_CARD);
1450 |
1451 |   /* FINALLY, PUNCH THE TRANSLATION RULE NUMBERS ASSOCIATED WITH
1452 |      THE LOOK-AHEAD--REDUCE STATES.                              */
1453 |
1454 |      CARD_IMAGE = '   DECLARE XLATION_RULE(' || #_LA_REDUCE_STATES - 1 ||
1455 |                                                  ') BIT(8) INITIAL(';
1456 |      DO STATE = 1 TO #_LA_REDUCE_STATES - 1;
1457 |         CALL PUNCH_CARD(XLATION_RULE(STATE) || ', ');
```

```
1458 |       END;
1459 |       CALL PUNCH_CARD(XLATION_RULE(STATE) || ');');
1460 |       CALL PUNCH_CARD(BLANK_CARD);
1461 |    END PUNCH_DPDA;     /*************************************************************/
1462 |
1463 |
1464 |
1465 | /*******************   MAIN PROGRAM --- SLR(1) ANALYSER   ***********************/
1466 |
1467 |MORE_GRAMMARS = TRUE;                                    /*  GLOBAL VARIABLE.  */
1468 |DO WHILE MORE_GRAMMARS;
1469 |
1470 |    CALL PRINT_DATE('SIMPLE LR(1) ANALYSER OF ', DATE_OF_GENERATION);
1471 |    CALL PRINT_DATE('TODAY IS ', DATE);
1472 |    ERROR_COUNT = 0;                                     /*  GLOBAL VARIABLE.  */
1473 |       CONTROL(@I) = FALSE;     /*  DO NOT LIST THE GRAMMAR CARDS.  */
1474 |       CONTROL(@G) = TRUE;      /*  DO LIST THE REFORMATTED GRAMMAR.  */
1475 |       CONTROL(@C) = FALSE;     /*  DO NOT LIST CONFIGURATION SETS.  */
1476 |       CONTROL(@F) = TRUE;      /*  DO LIST THE CHARACTERISTIC FSM.  */
1477 |       CONTROL(@L) = TRUE;      /*  DO LIST THE LOOK-AHEAD SETS.  */
1478 |       CONTROL(@P) = TRUE;      /*  DO PUNCH THE DPDA.  */
1479 |    FIRST_TIME, LAST_TIME = TIME;                        /*  GLOBAL VARIABLES.  */
1480 |    OUTPUT = 'START READING THE GRAMMAR.';   CALL PRINT_TIME;
1481 |    CALL READ_G;
1482 |    OUTPUT = 'THE GRAMMAR HAS BEEN READ.';   CALL PRINT_TIME;
1483 |
1484 |    IF ERROR_COUNT ¬= 0 THEN                                              DO;
1485 |       DECLARE STR CHARACTER;
1486 |       IF ERROR_COUNT = 1 THEN STR = 'THERE WAS ONE ERROR, ';
1487 |                          ELSE STR = 'THERE WERE ' || ERROR_COUNT || ' ERRORS, ';
1488 |       OUTPUT = STR || 'THEREFORE THE GRAMMAR WILL NOT BE ANALYSED.';
1489 |       OUTPUT = 'THE GRAMMAR IS LISTED BELOW FOR DEBUGGING PURPOSES.';
1490 |       DOUBLE_SPACE;                                                      END;
1491 |
1492 |    IF CONTROL(@G) | ERROR_COUNT ¬= 0 THEN                                 DO;
1493 |       EJECT_PAGE;
1494 |       CALL PRINT_G;                                                      END;
1495 |
1496 |    IF ERROR_COUNT = 0 THEN                                               DO;
1497 |       OUTPUT = 'START CONSTRUCTING THE GRAMMAR''S CHARACTERISTIC FSM.';
1498 |       CALL PRINT_TIME;
1499 |       IS_LR0, IS_SLR1 = TRUE;
1500 |       CALL COMPUTE_CFSM;
1501 |       OUTPUT = 'THE CFSM FOR THE GRAMMAR HAS BEEN COMPUTED.';
1502 |       CALL PRINT_TIME;
1503 |       EJECT_PAGE;
1504 |       IF CONTROL(@F) | ERROR_COUNT ¬= 0 THEN CALL PRINT_CFSM;              END;
1505 |
1506 |    IF ERROR_COUNT ¬ = 0
1507 |    THEN OUTPUT = 'SINCE THE CFSM IS IN ERROR, ANALYSIS WILL BE TERMINATED.';
1508 |    ELSE                                                                  DO;
1509 |       IF IS_LR0
1510 |       THEN OUTPUT = 'SURPRISE   THE GRAMMAR IS LR(0).';
1511 |       ELSE                                                               DO;
1512 |          OUTPUT = 'THE GRAMMAR IS NOT LR(0).  LOOK-AHEAD MUST BE ADDED.';
1513 |          DOUBLE_SPACE;
1514 |          OUTPUT = 'START COMPUTING LOOK-AHEAD SETS.';   CALL PRINT_TIME;
1515 |          CALL COMPUTE_SLR1_LA_SETS;
1516 |          OUTPUT = 'LOOK-AHEAD SETS HAVE BEEN COMPUTED.'; CALL PRINT_TIME;
1517 |          IF IS_SLR1 THEN OUTPUT = 'THE GRAMMAR IS SLR(1).';
1518 |          DOUBLE_SPACE;
```

```
1519 |          IF CONTROL(@L) | ¬IS_SLR1 | ERROR_COUNT ¬= 0 THEN                    |
1520 |             CALL PRINT_SLR1_LA_SETS;                                          |
1521 |          IF CONTROL(@P) & ERROR_COUNT = 0                                     |
1522 |          THEN                                                          DO;    |
1523 |             OUTPUT = 'START PUNCHING THE DPDA.';    CALL PRINT_TIME;         |
1524 |             CALL PUNCH_DPDA;                        CALL PRINT_TIME;  END;   |
1525 |          ELSE                                                                 |
1526 |             IF ERROR_COUNT ¬= 0 THEN                                          |
1527 |             OUTPUT = 'SINCE THE LOOK-AHEAD SETS ARE IN ERROR, EXECUTION ' ||  |
1528 |                      'WILL BE TERMINATED.';                            END;END;|
1529 |    OUTPUT = 'ALL DONE.';                                                      |
1530 |    EJECT_PAGE;                                                                |
1531 |END;     /*************************************************************************/|
1532 |EOF EOF EOF EOF EOF EOF
```

Some page-ejects have been eliminated (by cutting and pasting) for conciseness in this report.

```
*  FILE CONTROL BLOCK  22800    56000    57    140    400    188    180
*  LOAD FILE WRITTEN.
END OF COMPILATION SEPTEMBER 23, 1970.  CLOCK TIME = 11:7:25.24.

1532 CARDS CONTAINING 737 STATEMENTS WERE COMPILED.
NO ERRORS WERE DETECTED.
22588 BYTES OF PROGRAM, 47681 OF DATA, 2236 OF DESCRIPTORS, 5860 OF STRINGS.  TOTAL CORE
                                                          REQUIREMENT 78365 BYTES.

TOTAL TIME IN COMPILER   0:3:51.74.
SET UP TIME              0:0:16.37.
ACTUAL COMPILATION TIME  0:3:14.17.
POST-COMPILATION TIME    0:0:21.20.
COMPILATION RATE: 473 CARDS PER MINUTE.
```

SIMPLE LR(1) ANALYSER OF SEPTEMBER 22, 1970.

TODAY IS SEPTEMBER 22, 1970.

START READING THE GRAMMAR.
TIME USED SINCE THE LAST TIME PRINTOUT IS 0.01 SECONDS.
TOTAL TIME USED SO FAR IS 0.01 SECONDS.

@CONFIGURATION SETS
E E + T
T
T P ** T
P
P I
(E)

NO EXPLICIT GOAL SYMBOL WAS FOUND. E WILL BE USED AS THE GOAL SYMBOL.

THE GRAMMAR HAS BEEN READ.
TIME USED SINCE THE LAST TIME PRINTOUT IS 1.40 SECONDS.
TOTAL TIME USED SO FAR IS 1.41 SECONDS.

THE VOCABULARY

	TERMINAL SYMBOLS	NONTERMINALS
1	_I_	E
2	+	T
3	**	P
4	I	
5	(	
6	)	

E IS THE GOAL SYMBOL.

THE PRODUCTIONS

```
1   E  ::=   E + T
2        |   T

3   T  ::=   P ** T
4        |   P

5   P  ::=   I
6        |   ( E )
```

SOME STATISTICS ON THE GRAMMAR:

NUMBER OF TERMINAL SYMBOLS = 6
NUMBER OF NONTERMINAL SYMBOLS = 3
TOTAL NUMBER OF SYMBOLS = 9

NUMBER OF PRODUCTIONS = 6
SPACE REQUIRED TO STORE THE PRODUCTIONS = 62 BYTES, NOT INCLUDING THE VOCABULARY.
THE AVERAGE LENGTH OF THE RIGHT PARTS OF PRODUCTIONS = 2.14 SYMBOLS.

```
START CONSTRUCTING THE GRAMMAR'S CHARACTERISTIC FSM.
TIME USED SINCE THE LAST TIME PRINTOUT IS 0.45 SECONDS.
TOTAL TIME USED SO FAR IS 1.86 SECONDS.

THE CONFIGURATION SET FOR STATE 1 IS:
     1   <SYSTEM_GS>  ->   . _|_ F _|_

THE CONFIGURATION SET FOR STATE 2 IS:
     1   <SYSTEM_GS>  ->   _|_ . F _|_
     2   E  ->   . F + T
     3   E  ->   . T
     4   T  ->   . P ** T
     5   T  ->   . P
     6   P  ->   . I
     7   P  ->   . ( E )

THE CONFIGURATION SET FOR STATE 3 IS:
     1   <SYSTEM_GS>  ->   _|_ F . _|_
     2   E  ->   F . + T

THE CONFIGURATION SET FOR STATE 4 IS:
     1   T  ->   P .
     2   T  ->   P . ** T

THE CONFIGURATION SET FOR STATE 5 IS:
     1   P  ->   ( . E )
     2   E  ->   . E + T
     3   E  ->   . T
     4   T  ->   . P ** T
     5   T  ->   . P
     6   P  ->   . I
     7   P  ->   . ( E )

THE CONFIGURATION SET FOR STATE 6 IS:
     1   E  ->   E + . T
     2   T  ->   . P ** T
     3   T  ->   . P
     4   P  ->   . I
     5   P  ->   . ( E )

THE CONFIGURATION SET FOR STATE 7 IS:
     1   T  ->   P ** . T
     2   T  ->   . P ** T
     3   T  ->   . P
     4   P  ->   . I
     5   P  ->   . ( E )

THE CONFIGURATION SET FOR STATE 8 IS:
     1   P  ->   ( E . )
     2   E  ->   E . + T
```

THE CFSM FOR THE GRAMMAR HAS BEEN COMPUTED.
TIME USED SINCE THE LAST TIME PRINTOUT IS 5.45 SECONDS.
TOTAL TIME USED SO FAR IS 7.31 SECONDS.

THE CFSM FOR THE GRAMMAR IS AS FOLLOWS:

T H E L O O K - A H E A D -- R E D U C E T R A N S I T I O N

LOOK-AHEAD REDUCE STATE	DEFAULT TRANSITION TO STATE	NUMBER OF STATES TO POP	SYMBOL TO READ
1		0	F
2		0	P
3	*4	0	T
4		2	E
5		2	T
6		2	P

T H E R E A D T R A N S I T I O N S

READ STATES	SYMBOLS _1_	+	**	I	(	)	E	T	P
*1	*2								
*2				2	*5		*3	1	3
*3	EXIT	*6							
*4			*7						
*5				2	*5		*8	1	3
*6				2	*5			4	3
*7				2	*5			5	3
*8		*6				6			

T H E S Y M B O L S B E F O R E T H E S T A T E S

LA-REDUCE STATE	SYMBOL BEFORE THE STATE	READ STATE	SYMBOL BEFORE THE STATE
1	T	*1	ERROR_TOKEN
2	I	*2	_1_
3	P	*3	E
4	T	*4	P
5	T	*5	(
6	)	*6	+
		*7	**
		*8	F

```
SOME STATISTICS ON THE CFSM:

   NUMBER OF LOOK-AHEAD--REDUCE STATES = 6
   NUMBER OF READ STATES = 8
      TOTAL NUMBER OF STATES = 14

   SPACE REQUIRED FOR LOOK-AHEAD--REDUCE TRANSITIONS = 24 BYTES.
   SPACE REQUIRED FOR LOOK-AHEAD SETS = 3 BYTES.
   SPACE REQUIRED FOR READ TRANSITIONS = 80 BYTES.
      TOTAL SPACE REQUIRED FOR THE CFSM = 107 BYTES, NOT INCLUDING THE VOCABULARY.


THE GRAMMAR IS NOT LR(0).  LOOK-AHEAD MUST BE ADDED.


START COMPUTING LOOK-AHEAD SETS.
TIME USED SINCE THE LAST TIME PRINTOUT IS 0.82 SECONDS.
TOTAL TIME USED SO FAR IS 8.13 SECONDS.


LOOK-AHEAD SETS HAVE BEEN COMPUTED.
TIME USED SINCE THE LAST TIME PRINTOUT IS 0.23 SECONDS.
TOTAL TIME USED SO FAR IS 8.36 SECONDS.


THE GRAMMAR IS SLR(1).

      T H E   L O O K - A H E A D   S E T S


NONTERMINAL      TERMINAL SYMBOLS
  SYMBOLS      1   3   5
                 2   4   6
             ------------
          E | 1 1         1
          T | 1 1         1
          P | 1 1 1     . 1


START PUNCHING THE DPDA.
TIME USED SINCE THE LAST TIME PRINTOUT IS 0.20 SECONDS.
TOTAL TIME USED SO FAR IS 8.56 SECONDS.


TIME USED SINCE THE LAST TIME PRINTOUT IS 1.52 SECONDS.
TOTAL TIME USED SO FAR IS 10.08 SECONDS.


ALL DONE.
```

APPENDIX III

```
X P L  COMPILATION  - U OF C AT SANTA CRUZ - XCOM  III VERSION OF APRIL 27, 1970.  CLOCK

TODAY IS SEPTEMBER 22, 1970.  CLOCK TIME = 9:41:28.23.

  1 | /*************************************************************************|
  2 |                                                                         |
  3 |                          A SIMPLE LR(1) SKELETON                        |
  4 |                                                                         |
  5 |                                BY                                       |
  6 |                                                                         |
  7 |                          FRANKLIN L. DE REMER                           |
  8 |                                                                         |
  9 |                          UNIVERSITY OF CALIFORNIA                       |
 10 |                            SANTA CRUZ, CALIFORNIA                       |
 11 |                              AUGUST 31, 1970                            |
 12 |                                                                         |
 13 |                                                                         |
 14 |                                PREFACE                                  |
 15 |                                                                         |
 16 |     THE FOLLOWING PROGRAM IS A SKELETAL COMPILER DESIGNED AROUND THE USE OF
 17 |SIMPLE LR(K) PARSING TECHNIQUES.  IT CONSISTS PRIMARILY OF A PARSING PROCEDURE,
 18 |DPDA_PARSER, WHICH USES TABLES GENERATED FROM A GRAMMAR BY OUR SIMPLE LR(K)
 19 |GRAMMAR ANALYSER.  NOTE THAT WE HAVE NOT AS YET WRITTEN AN ERROR-RECOVERY
 20 |ROUTINE FOR THE PARSER; PART OF WHAT IS NEEDED IS A NONDETERMINISTIC PUSHDOWN
 21 |AUTOMATON (NPDA).  ALSO, NOTE THAT WE HAVE NOT SPECIFIED EITHER THE LEXICAL
 22 |ANALYSER OR THE SYNTHESIZER (CODE GENERATOR) TO BE USED WITH THE PARSER SINCE
 23 |THESE ROUTINES ARE, OF COURSE, IRRELEVANT FOR OUR PURPOSES HERE; APPROPRIATE
 24 |SUCH ROUTINES MAY BE EASILY ADAPTED FROM THOSE IN THE XPL SYSTEM.
 25 |     IT SHOULD BE NOTED THAT WE HAVE USED THE XPL COMPILER TO ASCERTAIN THE
 26 |SYNTACTIC CORRECTNESS OF THIS PROGRAM, BUT WE HAVE NOT DEBUGGED A RUNNING
 27 |VERSION OF IT.
 28 |
 29 |*************************************************************************/
 30 |
 31 |
 32 |
 33 |   /*  FIRST SOME LANGUAGE EXTENTIONS.  */
 34 |
 35 |   DECLARE BOOLEAN       LITERALLY 'BIT(1)',
 36 |           TRUE          LITERALLY '1',
 37 |           FALSE         LITERALLY '0';
 38 |
 39 |
 40 |   /*   NEXT WE INITIALIZE THE GLOBAL CONSTANTS THAT DEPEND UPON THE INPUT
 41 |    GRAMMAR.  THE FOLLOWING CARDS ARE PUNCHED BY THE SLR(1) GRAMMAR ANALYZER. */
 42 |
 43 |   /*  THE TABLES BELOW WERE GENERATED FOR THE FOLLOWING GRAMMAR:
 44 |   E  ::=  E + T
 45 |         |  T
 46 |   T  ::=  P ** T
 47 |         |  P
 48 |   P  ::=  I
 49 |         |  ( E )
 50 |   END OF GRAMMAR.  */
 51 |
 52 |   DECLARE #_TERMINALS LITERALLY '6',
 53 |           #_NTS       LITERALLY '3',
 54 |           #_SYMS      LITERALLY '9';
```

```
55 |    DECLARE V(#_SYMS) CHARACTER INITIAL ('_|_', '+', '**', '|', '(', ')', 'F',
56 |       'T', 'P', 'ERROR_TOKEN');
57 |    DECLARE #_LA_REDUCE_STATES LITERALLY '6';
58 |    DECLARE #_BYTES_PER_NT LITERALLY '1';
59 |    DECLARE LOOK_AHEAD_TABLE(3) BIT(8) INITIAL (196, 199, 231, 0);
60 |    DECLARE #_TO_POP(5) BIT(8) INITIAL (0, 0, 0, 2, 2, 2);
61 |    DECLARE SYM_TO_READ(5) BIT(8) INITIAL (6, 8, 7, 6, 7, 8);
62 |    DECLARE DEFAULT_XITION(5) BIT(8) INITIAL (15, 15, 9, 15, 15, 15);
63 |    DECLARE #_READ_STATES LITERALLY '9',
64 |            #_LAR_PLUS_READ_STATES LITERALLY '14';
65 |    DECLARE READ_XITIONS_TABLE(72) BIT(8) INITIAL (7, 15, 15, 15, 15, 15, 15,
66 |       15, 15, 15, 15, 15, 1, 10, 15, 8, 0, 2, 14, 11, 15, 15, 15, 15, 15, 15,
67 |       15, 15, 15, 12, 15, 15, 15, 15, 15, 15, 15, 15, 15, 1, 10, 15, 13, 0, 2,
68 |       15, 15, 15, 1, 10, 15, 15, 3, 2, 15, 15, 15, 1, 10, 15, 15, 4, 2, 15, 11,
69 |       15, 15, 15, 5, 15, 15, 15, 0);
70 |    DECLARE SYM_BEFORE(15) BIT(8) INITIAL (7, 3, 8, 7, 7, 5, 9, 0, 6, 8, 4, 1,
71 |       2, 6, 3, 9);
72 |    DECLARE XLATION_RULE(5) BIT(8) INITIAL(2, 5, 0, 1, 3, 6);
73 |
74 |    /*  END OF CARDS PUNCHED BY THE GRAMMAR ANALYSER.                    */
75 |
76 |
77 |
78 |
79 |SCAN:    /*  A LEXICAL SCANNER.  */
80 |    PROCEDURE BIT(8);
81 |       DECLARE NEXT_SYMBOL BIT(8);
82 |       RETURN NEXT_SYMBOL;    /*  TEMPORARY.  */
83 |    END SCAN;
84 |
85 |
86 |
87 |
88 |SYNTHESIZE:    /*  A CODE GENERATOR.  */
89 |    PROCEDURE (XLATION_RULE_#);
90 |       DECLARE XLATION_RULE_# BIT(8);
91 |       OUTPUT = 'PRODUCTION NUMBER = ' || XLATION_RULE_#;
92 |       RETURN;    /*  TEMPORARY.  */
93 |    END SYNTHESIZE;
94 |
95 |
96 |
97 |
98 |DPDA_PARSER:
99 |    PROCEDURE;
100|
101|       DECLARE STATE_STACK(127) BIT(8), SP BIT(7),
102|           /*  SP POINTS TO THE ELEMENT ABOVE THE TOP-OF-STACK.  */
103|               CURRENT_STATE LITERALLY 'STATE_STACK(SP)',
104|               PUSH_CURRENT_STATE_ON_STATE_STACK LITERALLY
105|                  'IF SP < 127 THEN SP = SP + 1; -
106|                   ELSE CALL ERROR_RECOVERY(''PARSE-STACK OVERFLOW.'')',
107|               PREV_STATE LITERALLY 'STATE_STACK(SP - 1)',
108|               NEXT_STATE LITERALLY 'STATE_STACK(SP + 1)',
109|               READ LITERALLY 'TRUE',
110|               LOOK_AHEAD LITERALLY 'FALSE';
111|
112|       DECLARE IS_LA_REDUCE LITERALLY '#_LA_REDUCE_STATES >',
113|               IS_READ      LITERALLY '#_LAR_PLUS_READ_STATES >',
114|               IS_EXIT      LITERALLY '#_LAR_PLUS_READ_STATES =',
115|               START_STATE  LITERALLY '#_LA_REDUCE_STATES';
```

REFERENCES

Che 67 — Cheatham, T. E., Jr. The Theory and Construction of Compilers. Massachusetts Computer Associates, Inc., Wakefield, Mass., 1967.

DeR 69 — DeRemer, F. L. Practical Translators for LR(k) Languages. Ph.D. thesis. Mass. Inst. of Tech. Cambridge, Mass. Sept., 1969.

DeR 70a — _____. Minimal LR(k) Parsers. To be published.

DeR 70b — _____. Error Recovery using LR(k) Techniques. To be published.

Ear 70 — Earley, J. An Efficient Context-Free Parsing Algorithm, CACM 13 (February 1970) pp. 94-102.

Flo 64 — Floyd, R. W. Bounded context syntactic analysis. CACM 7 (February 1964) pp. 62-67.

H&U 69 — Hopcroft, J. E., and Ullman, J. D. Formal Languages and their Relation to Automata, Addison-Wesley, Inc., Reading, Mass., 1969.

I&M 70 — Ichbiah, J. D., and Morse. A Technique for Generating Almost Optimal Floyd-Evans Productions for Precedence Grammars. Comm. ACM 13 (8), August 1970.

Knu 65 — Knuth, D. E. On the translation of languages from left to right. Inf. Contr. 8 (October 1965) pp. 607-639.

MHW 70 — McKeeman, W. M., Horning, J. J., and Wortman, D. B. A Compiler Generator. Prentice-Hall, Inc., Englewood Cliffs, N. J., 1970.

W&W 66 — Wirth, N. and Weber, H. EULER - a generalization of ALGOL, and its formal definition: parts I, II. CACM 9 (Jan., Feb., 1966) pp. 13-25, 89-99.

R&M 69 — Reddy, D. R., and McKeeman, W. M. Computer Programming: An Introduction to PL, preliminary report, Stanford University, Stanford, California, 1969.

```
177 |                CALL SYNTHESIZE(XLATION_RULE(CURRENT_STATE));                    |
178 |                IF #_TO_POP(CURRENT_STATE) ¬= -1                                 |
179 |                THEN SP = SP - #_TO_POP(CURRENT_STATE);                          |
180 |                ELSE /*  SPECIAL CASE FOR EMPTY RIGHT PARTS.  */          DO:    |
181 |                     CURRENT_STATE = ULTIMATE_READ_STATE_OF(CURRENT_STATE);      |
182 |                     PUSH_CURRENT_STATE_ON_STATE_STACK;                  END;    |
183 |                CURRENT_STATE = READ_XITION(PREV_STATE, SYM);            END;    |
184 |             ELSE                                                              |
185 |                CURRENT_STATE = DEFAULT_XITION(CURRENT_STATE);           END;    |
186 |          ELSE                                                                 |
187 |                                                                               |
188 |          IF IS_READ(CURRENT_STATE) THEN                                  DO:    |
189 |             NEXT_STATE = READ_XITION(CURRENT_STATE, INPUT_QUEUE(READ));       |
190 |             PUSH_CURRENT_STATE_ON_STATE_STACK;                           END;    |
191 |          ELSE                                                                 |
192 |                                                                               |
193 |          IF IS_EXIT(CURRENT_STATE) THEN RETURN;                               |
194 |          ELSE  /*                                                             |
195 |                                                                               |
196 |          IF IS_ERROR(CURRENT_STATE) THEN  */                                  |
197 |             CALL ERROR_RECOVERY('ILLEGAL SYMBOL PAIR.');                      |
198 |       END;                                                                    |
199 |                                                                               |
200 |    END DPDA_PARSER;                                                           |
201 |                                                                               |
202 |                                                                               |
203 |                                                                               |
204 | /*****************************    MAIN  PROGRAM    ********************************/|
205 |                                                                               |
206 |    /*  SET UP FOR COMPILATION.  */                    /*  TEMPORARY.  */     |
207 |                                                                               |
208 |    CALL DPDA_PARSER;     /*  DRIVES BOTH SCAN AND SYNTHESIZE.  */             |
209 |                                                                               |
210 |    /*  PRINT SUMMARY INFORMATION, ETC.  */            /*  TEMPORARY.  */     |
211 |                                                                               |
212 | /*************************************************************************** */|
213 |EOF EOF EOF EOF
```

```
*  FILE CONTROL BLOCK  2400    2400    6    6    400    400    60
*  LOAD FILE WRITTEN.
END OF COMPILATION SEPTEMBER 22, 1970.  CLOCK TIME = 9:42:12.53.

213 CARDS CONTAINING 65 STATEMENTS WERE COMPILED.
NO ERRORS WERE DETECTED.
2398 BYTES OF PROGRAM, 1709 OF DATA, 84 OF DESCRIPTORS, 264 OF STRINGS.  TOTAL CORE REQUI


TOTAL TIME IN COMPILER    0:0:44.41.
SET UP TIME               0:0:16.48.
ACTUAL COMPILATION TIME   0:0:21.28.
POST-COMPILATION TIME     0:0:6.65.
COMPILATION RATE: 600 CARDS PER MINUTE.
```

```
116 |
117 |
118 |       READ_XITION:
119 |          PROCEDURE (STATE, SYM) BIT(8);
120 |             DECLARE (STATE, SYM) BIT(8);
121 |             RETURN READ_XITIONS_TABLE(#_SYMS*(STATE - #_LA_REDUCE_STATES)+ SYM);
122 |          END;
123 |
124 |
125 |          DECLARE MASK(7) BIT(8) INITIAL
126 |             ("(1)10000000", "(1)01000000", "(1)00100000", "(1)00010000",
127 |              "(1)00001000", "(1)00000100", "(1)00000010", "(1)00000001");
128 |       IS_IN_LOOK_AHEAD_SET:
129 |          PROCEDURE (STATE, SYM) BOOLEAN;
130 |             DECLARE (STATE, SYM) BIT(8);
131 |             DECLARE NT_SYM BIT(8);
132 |             NT_SYM = SYM_TO_READ(STATE) - #_TERMINALS;
133 |             RETURN (LOOK_AHEAD_TABLE(#_BYTES_PER_NT * NT_SYM + SHR(SYM, 3))
134 |                                                & MASK(SYM & "(1)111")) ¬= 0;
135 |          END;
136 |
137 |
138 |       ULTIMATE_READ_STATE_OF:
139 |          PROCEDURE (STATE) BIT(8);
140 |             DECLARE STATE BIT(8);
141 |             DO WHILE¬IS_READ(STATE);
142 |                STATE = DEFAULT_XITION(STATE);
143 |             END;
144 |             RETURN STATE;
145 |          END;
146 |
147 |
148 |       ERROR_RECOVERY:
149 |          PROCEDURE (MESSAGE);
150 |             DECLARE MESSAGE CHARACTER;
151 |             CALL EXIT;    /*  TEMPORARY.  */
152 |          END;
153 |
154 |
155 |          DECLARE QUEUE(3) BIT(8), QP BIT(2);
156 |       INPUT_QUEUE:
157 |          PROCEDURE (TO_READ) BIT(8);
158 |             DECLARE TO_READ BOOLEAN;
159 |             IF TO_READ THEN                                              DO;
160 |                QUEUE(QP) = SCAN;
161 |                QP = QP + 1 & "(1)11";                                    END;
162 |             RETURN QUEUE(QP);
163 |          END;
164 |
165 |
166 |    /* THE BODY OF THE PROCEDURE "DPDA_PARSER".  */
167 |
168 |       DO QP = 0 TO 3;   QUEUE(QP) = SCAN;   END;     QP = 0;
169 |       SP=0;   CURRENT_STATE = START_STATE;
170 |
171 |       DO WHILE TRUE;    /*  EXIT VIA A RETURN BY THE EXIT STATE.  */
172 |
173 |          IF IS_LA_REDUCE(CURRENT_STATE) THEN                            DO;
174 |             IF IS_IN_LOOK_AHEAD_SET(CURRENT_STATE, INPUT_QUEUE(LOOK_AHEAD))
175 |             THEN                                                        DO;
176 |                DECLARE SYM BIT(8);   SYM = SYM_TO_READ(CURRENT_STATE);
```