The Earley Parsing Algorithm

Dustin Mitchell

March 26, 2001

1 Introduction

Most commercially and freely available parser implementations can handle only a subset of all context-free grammars. In practice, these implementations can parse almost all practical programming languages. However, they often require changes to the grammar to bring it into the handled subset. These changes do not alter the set of sentences recognized by the grammar, but they obscure the structure of the grammar to the human reader, and complicate production of a useful parse tree.

For implementers of a well-specified language, the common parser implementations provide very efficient parsing in exchange for some hard work constructing an acceptable context-free grammar. Those implementing an ad-hoc language, in which the syntax is changing while the implementation is being written, need to work with a much more maintainable representation for the grammar. Further, in the process of prototyping new features in the language, the developer may wish to use an ambiguous or otherwise complicated grammar. There is a need for a parser which can handle arbitrary grammars, but which performs reasonably well on most grammars.

Jay Earley, in his 1968 PhD thesis [?], presented an algorithm which he claimed was capable of doing just this. This paper will describe Earley's algorithm and the author's implementation of it, and will offer commentary on its practicality for parsing experimental languages.

2 The Algorithm

At its simplest, Earley's algorithm functions as a recognizer. This section presents an informal discussion of this aspect of the algorithm. For a more precise description, see [?]. Although the original algorithm operates with a k-token 'lookahead', we assume $k = 0$, eliminating any reference to the lookahead.

2.1 The Recognizer

The recognizer scans a set of tokens $X_0, \ldots, X_{n-1}$ from left to right. For each token X_i, it constructs a *state set* S_i representing the collection of possible states of the recognition process at position i. Each of these states is given as a position within a rule from the grammar, with a reference to the token position at which processing of the rule was begun.

Definition 1 *A* ***state*** *has the form*

$$\alpha \xrightarrow{p} \beta.\gamma$$

where α is a nonterminal and β and γ are (possibly empty) strings of grammar symbols, and $\alpha \longrightarrow \beta\gamma$ is a rule of the grammar.

The dot represents the parser's position within the rule, and p represents the beginning token position.

Definition 2 *A state is* ***final*** *if it has no symbols to the right of its dot. The* ***next symbol*** *of a non-final state is the symbol to the right of the dot. A state is* ***terminal*** *if it is not final and its next symbol is a terminal. A state is* ***nonterminal*** *if it is not final and its next symbol is a nonterminal.*

The recognizer is initialized with the special state $\mathbf{S}' \xrightarrow{0} .\mathbf{S} \dashv \in S_0$, where $\mathbf{S}'$ is a unique start symbol and $\dashv$ represents end-of-file. It is assumed that $X_i = \dashv$ for $i \geq n$. Processing then proceeds state-by-state, through successive state sets. For a state $s \in S_i$, one of three actions are possible:

Scan: If s is terminal, and its next symbol is $\mathbf{X_i}$, then the algorithm adds a copy of s to S_{i+1} with the dot moved one symbol to the right.

Predict: If s is nonterminal, and its next symbol is $\mathbf{N}$, then for each rule $\mathbf{N} \longrightarrow \alpha$ in the grammar, the state $\mathbf{N} \xrightarrow{i} .\alpha$ is added to S_i.

Complete: If s is final, then for each 'parent' state in S_p with the dot immediately preceding $\mathbf{N}$, the algorithm adds a similar state to S_i, with the dot moved to the right of the $\mathbf{N}$.

The input string is recognized if $S_{n+1} = \{\mathbf{S}' \xrightarrow{0} \mathbf{S} \dashv .\}$ and all other states have been processed. If no states remain to be processed or if S_{n+1} contains more states than this one, then the string is not recognized.

2.2 The Parser

With only a slight modification, the recognizer can create a parse tree for any recognized string. Each state is extended to contain a list $[\alpha_1, \alpha_2, \ldots]$ containing the parse trees for each of the grammar symbols to the left of the dot. The parser is initialized by putting $\mathbf{S}' \xrightarrow{0} .\mathbf{S} \dashv []$ into S_0. Processing then proceeds as above, with the following modifications:

Scan: Append X_i to the list for the new state.

Predict: Create each new state with an empty list.

Complete: Each new state carries a copy of the parent state's list, to which is appended the list for s as a single element.

3 Examples

3.1 A Right-associative Grammar

$$\begin{array}{lll} \mathbf{E} \longrightarrow \mathbf{T} & \mathbf{T} \longrightarrow \mathbf{F} & \mathbf{F} \longrightarrow \mathbf{IDENT} \\ \mathbf{E} \longrightarrow \mathbf{T} + \mathbf{E} & \mathbf{T} \longrightarrow \mathbf{F} * \mathbf{T} & \end{array}$$

Table 1: Grammar EXP − R

Given grammar EXP − R (Table 1) The recognizer produces the states shown in Table 2 on input '$\mathbf{A} * \mathbf{B} + \mathbf{C}$'. States are grouped into state sets with horizontal lines.

State set	State
$S_0 =$	$\mathbf{S'} \xrightarrow{0} .\mathbf{E} \dashv$
	$\mathbf{E} \xrightarrow{0} .\mathbf{T}$
	$\mathbf{E} \xrightarrow{0} .\mathbf{T} + \mathbf{E}$
	$\mathbf{T} \xrightarrow{0} .\mathbf{F}$
	$\mathbf{T} \xrightarrow{0} .\mathbf{F} * \mathbf{T}$
	$\mathbf{F} \xrightarrow{0} .\mathbf{IDENT}$
$S_1 =$	$\mathbf{F} \xrightarrow{0} \mathbf{IDENT}.$
	$\mathbf{T} \xrightarrow{0} \mathbf{F}.$
	$\mathbf{T} \xrightarrow{0} \mathbf{F}. * \mathbf{T}$
	$\mathbf{E} \xrightarrow{0} \mathbf{T}.$
	$\mathbf{E} \xrightarrow{0} \mathbf{T}. + \mathbf{E}$
	$\mathbf{S'} \xrightarrow{0} \mathbf{E}. \dashv$
$S_2 =$	$\mathbf{T} \xrightarrow{0} \mathbf{F} * .\mathbf{T}$
	$\mathbf{T} \xrightarrow{2} .\mathbf{F}$
	$\mathbf{T} \xrightarrow{2} .\mathbf{F} * \mathbf{T}$
	$\mathbf{F} \xrightarrow{2} .\mathbf{IDENT}$
$S_3 =$	$\mathbf{F} \xrightarrow{2} \mathbf{IDENT}.$
	$\mathbf{T} \xrightarrow{2} \mathbf{F}.$
	$\mathbf{T} \xrightarrow{2} \mathbf{F}. * \mathbf{T}$
	$\mathbf{T} \xrightarrow{0} \mathbf{F} * \mathbf{T}.$
	$\mathbf{E} \xrightarrow{0} \mathbf{T}.$
	$\mathbf{E} \xrightarrow{0} \mathbf{T}. + \mathbf{E}$
	$\mathbf{S'} \xrightarrow{0} \mathbf{E}. \dashv$
$S_4 =$	$\mathbf{E} \xrightarrow{0} \mathbf{T} + .\mathbf{E}$
	$\mathbf{E} \xrightarrow{4} .\mathbf{T}$
	$\mathbf{E} \xrightarrow{4} .\mathbf{T} + \mathbf{E}$
	$\mathbf{T} \xrightarrow{4} .\mathbf{F}$
	$\mathbf{T} \xrightarrow{4} .\mathbf{F} * \mathbf{T}$
	$\mathbf{F} \xrightarrow{4} .\mathbf{IDENT}$
$S_5 =$	$\mathbf{F} \xrightarrow{4} \mathbf{IDENT}.$
	$\mathbf{T} \xrightarrow{4} \mathbf{F}.$
	$\mathbf{T} \xrightarrow{4} \mathbf{F}. * \mathbf{T}$
	$\mathbf{E} \xrightarrow{4} \mathbf{T}.$
	$\mathbf{E} \xrightarrow{4} \mathbf{T}. + \mathbf{E}$
	$\mathbf{E} \xrightarrow{0} \mathbf{T} + \mathbf{E}.$
	$\mathbf{S'} \xrightarrow{0} \mathbf{E}. \dashv$
$S_6 =$	$\mathbf{S'} \xrightarrow{0} \mathbf{E} \dashv .$

Table 2: States processed while parsing $\mathbf{A} * \mathbf{B} + \mathbf{C}$ under grammar EXP − R

Beginning with $\mathbf{S'} \xrightarrow{0} .\mathbf{E} \dashv$, the recognizer predicts

$$\mathbf{E} \xrightarrow{0} .\mathbf{T}$$
$$\mathbf{E} \xrightarrow{0} .\mathbf{T} + \mathbf{E}$$

and the both of these cause the prediction of

$$\mathbf{T} \xrightarrow{0} .\mathbf{F}$$
$$\mathbf{T} \xrightarrow{0} .\mathbf{F} * \mathbf{T}$$

which further causes the prediction of

$$\mathbf{F} \xrightarrow{0} .\mathbf{IDENT}$$

Finally, this state scans the first token, an identifier, producing

$$\mathbf{F} \xrightarrow{0} \mathbf{IDENT}.$$

in S_1. This state is completed, producing

$$\mathbf{T} \xrightarrow{0} \mathbf{F}.$$
$$\mathbf{T} \xrightarrow{0} \mathbf{F}. * \mathbf{T}$$

The first of these is further completed, resulting in

$$\mathbf{E} \xrightarrow{0} \mathbf{T}.$$
$$\mathbf{E} \xrightarrow{0} \mathbf{T}. + \mathbf{E}$$

and the first of these is once more completed, giving

$$\mathbf{S}' \xrightarrow{0} \mathbf{E}. \dashv$$

Only one state, $\mathbf{T} \xrightarrow{0} \mathbf{F}. * \mathbf{T}$, scans, giving

$$\mathbf{T} \xrightarrow{0} \mathbf{F} * .\mathbf{T}$$

which results in the prediction of

$$\mathbf{T} \xrightarrow{2} .\mathbf{F}$$
$$\mathbf{T} \xrightarrow{2} .\mathbf{F} * \mathbf{T}$$
$$\mathbf{F} \xrightarrow{2} .\mathbf{IDENT}$$

all of which were created in token position 2. Processing continues in this manner, until S_6 is produced, containing the single state

$$\mathbf{S}' \xrightarrow{0} \mathbf{E} \dashv .$$

and the string is recognized.

3.2 Comparison of two grammars

$\mathbf{E} \longrightarrow \mathbf{T}$	$\mathbf{T} \longrightarrow \mathbf{F}$	$\mathbf{F} \longrightarrow \mathbf{IDENT}$
$\mathbf{E} \longrightarrow \mathbf{E} + \mathbf{T}$	$\mathbf{T} \longrightarrow \mathbf{T} * \mathbf{F}$	

Table 3: Grammar EXP − L

Consider grammar EXP − L in Table 3, a left-associative grammar describing the same language as EXP − R. Table 4 shows the recognizer's states in parsing '$\mathbf{A} + \mathbf{B} + \mathbf{C} + \mathbf{D}$' with each grammar.

Note that the the number of states in each state set is relatively constant for the left-associative grammar. On the other hand, S_7 is quite large when recognizing under EXP − R. In particular, observe states

$$\mathbf{E} \xrightarrow{6} \mathbf{T}.$$
$$\mathbf{E} \xrightarrow{4} \mathbf{T} + \mathbf{E}.$$
$$\mathbf{E} \xrightarrow{2} \mathbf{T} + \mathbf{E}.$$
$$\mathbf{E} \xrightarrow{0} \mathbf{T} + \mathbf{E}.$$

which represent the completion of each of the nested applications of the rule $\mathbf{E} \longrightarrow \mathbf{T} + \mathbf{E}$ which all finish on the last token position. If the input string had contained more summands, then state set S_n would contain additional completions for the additional summations.

Grammar EXP − L is a *bounded state* algorithm, because the size of a given state is bounded by some constant. On the other hand, grammar EXP − R is not bounded state, because for input string $A_1 + \cdots + A_m$, state set $2m - 1$ would contain at least $m - 1$ states.

Although the algorithm correctly recognizes sentences using either grammar, it does so more slowly for EXP − R, because it is not bounded-state. This result has been confirmed with empirical tests using large input strings.

3.3 Parsing an ambiguous grammar

Grammar AMB (Table 5) is an ambiguous grammar. Table 6 shows the results of parsing the string '$A+B+C$' using this grammar. Note that S_6 contains two final states, one for each of the possible parses. The Earley algorithm handles ambiguous grammars without error, and presents its results in a clear manner.

4 Implementation

The Earley parsing algorithm was implemented in Python by John Aycock [?]. The algorithm complements Python's utility as a rapid application design language, but its speed is inadequate, particularly since Python is an interpreted language.

The author re-implemented the algorithm in C, in the form of a compiler-compiler. The compiler-compiler is written in Python. Unlike parser generators for LALR grammars such as Yacc or Bison, there is very little work that can be done at compile time for an Earley parser, so the compiler-compiler has little else to do than encode the grammar rules numerically and output C code for the algorithm.

The C implementation uses DOH, a simple dynamically typed object library developed by David Beazley [?]. C functions may be associated with grammar rules. Those functions are given a DOH list of parse trees for the symbols on the right hand side of the rule, and are expected to return a parse tree for the rule itself, in the form of another DOH object.

When a state is added to a state set, the implementation must avoid duplicating a state already in the state set. Although this can be done with a search of the state set, a few simple observations lead to a more efficient strategy. First, any state added by scanning will be unique, so the implementation need not search for duplicates. Second, any state added by completion will be unique, assuming that the action functions produce distinct outputs on distinct inputs. Third, prediction for a given nonterminal need only happen once per state set.

The implementation, then, keeps a list of symbols predicted for the current state set, and searches this list before predicting to avoid re-predicting a symbol. No further checking is performed for duplicate states.

5 Time and space bounds

The algorithm's running time is, in general, $O(n^3)$. On unambiguous grammars and certain ambiguous grammars, it is $O(n^2)$. On a relatively large set of grammars, including LR(0) and bounded-state grammars (See section 3.2), it is $O(n)$. The constants on each of these terms, however, is much larger than for comparable algorithms such as that implemented by Yacc and Bison for LALR grammars.

Space consumption by the algorithm is $O(n^2)$, again with a considerably large constant.

See [?, ?] for a formal discussion of the performance characteristics of the algorithm.

6 Conclusion

Earley's parsing algorithm is extremely useful in prototyping new languages, as it can parse using any grammar, including ambiguous grammars. Originally published in 1968, it is not a well-known algorithm, probably because it cannot match the efficiency of more specialized, restrictive parsing algorithms. However, as rapid application development becomes more popular, the algorithm may prove extremely useful to designers who wish to add sophisticated languages to their applications.

The algorithm does have some drawbacks. First, because the algorithm maintains all possible parser states simultaneously, it is not currently possible to maintain any sort of state which changes as the parser moves through the tokens. This is a particularly acute problem when parsing C or C++, where identifiers must be classified by the parser as type names or variable names. Conventional parsers allow extraction of type names from type declarations during the parse, and make use of this information on encountering unknown identifiers later in the parse. Earley parsers cannot collect such methods, and so must resort to heuristics to differentiate type names from variable names when parsing C.

Second, unlike Yacc or Bison, the Earley algorithm gives no information as to the classification of the grammar it uses. A language designer may develop a grammar which parses all test strings unambiguously, but is nonetheless ambiguous.

With further research, both of these drawbacks could be overcome. The algorithm is flexible and useful, its performance is acceptable on most practical grammars, and it performs correctly on all grammars.

EXP − L

$s_0 =$			
	S'	$\xrightarrow{0}$	$.E \dashv$
	E	$\xrightarrow{0}$	$.T$
	E	$\xrightarrow{0}$	$.E + T$
	T	$\xrightarrow{0}$	$.F$
	T	$\xrightarrow{0}$	$.T * F$
	F	$\xrightarrow{0}$	$.IDENT$
$s_1 =$			
	F	$\xrightarrow{0}$	$IDENT.$
	T	$\xrightarrow{0}$	$F.$
	E	$\xrightarrow{0}$	$T.$
	T	$\xrightarrow{0}$	$T. * F$
	S'	$\xrightarrow{0}$	$E. \dashv$
	E	$\xrightarrow{0}$	$E. + T$
$s_2 =$			
	E	$\xrightarrow{0}$	$E + .T$
	T	$\xrightarrow{2}$	$.F$
	T	$\xrightarrow{2}$	$.T * F$
	F	$\xrightarrow{2}$	$.IDENT$
$s_3 =$			
	F	$\xrightarrow{2}$	$IDENT.$
	T	$\xrightarrow{2}$	$F.$
	E	$\xrightarrow{0}$	$E + T.$
	T	$\xrightarrow{2}$	$T. * F$
	S'	$\xrightarrow{0}$	$E. \dashv$
	E	$\xrightarrow{0}$	$E. + T$
$s_4 =$			
	E	$\xrightarrow{0}$	$E + .T$
	T	$\xrightarrow{4}$	$.F$
	T	$\xrightarrow{4}$	$.T * F$
	F	$\xrightarrow{4}$	$.IDENT$
$s_5 =$			
	F	$\xrightarrow{4}$	$IDENT.$
	T	$\xrightarrow{4}$	$F.$
	E	$\xrightarrow{0}$	$E + T.$
	T	$\xrightarrow{4}$	$T. * F$
	S'	$\xrightarrow{0}$	$E. \dashv$
	E	$\xrightarrow{0}$	$E. + T$
$s_6 =$			
	E	$\xrightarrow{0}$	$E + .T$
	T	$\xrightarrow{6}$	$.F$
	T	$\xrightarrow{6}$	$.T * F$
	F	$\xrightarrow{6}$	$.IDENT$
$s_7 =$			
	F	$\xrightarrow{6}$	$IDENT.$
	T	$\xrightarrow{6}$	$F.$
	E	$\xrightarrow{0}$	$E + T.$
	T	$\xrightarrow{6}$	$T. * F$
	S'	$\xrightarrow{0}$	$E. \dashv$
	E	$\xrightarrow{0}$	$E. + T$
$s_8 =$			
	S'	$\xrightarrow{0}$	$E \dashv .$

EXP − R

$s_0 =$			
	S'	$\xrightarrow{0}$	$.E \dashv$
	E	$\xrightarrow{0}$	$.T$
	E	$\xrightarrow{0}$	$.T + E$
	T	$\xrightarrow{0}$	$.F$
	T	$\xrightarrow{0}$	$.F * T$
	F	$\xrightarrow{0}$	$.IDENT$
$s_1 =$			
	F	$\xrightarrow{0}$	$IDENT.$
	T	$\xrightarrow{0}$	$F.$
	T	$\xrightarrow{0}$	$F. * T$
	E	$\xrightarrow{0}$	$T.$
	E	$\xrightarrow{0}$	$T. + E$
	S'	$\xrightarrow{0}$	$E. \dashv$
$s_2 =$			
	E	$\xrightarrow{0}$	$T + .E$
	E	$\xrightarrow{2}$	$.T$
	E	$\xrightarrow{2}$	$.T + E$
	T	$\xrightarrow{2}$	$.F$
	T	$\xrightarrow{2}$	$.F * T$
	F	$\xrightarrow{2}$	$.IDENT$
$s_3 =$			
	F	$\xrightarrow{2}$	$IDENT.$
	T	$\xrightarrow{2}$	$F.$
	T	$\xrightarrow{2}$	$F. * T$
	E	$\xrightarrow{2}$	$T.$
	E	$\xrightarrow{2}$	$T. + E$
	E	$\xrightarrow{0}$	$T + E.$
	S'	$\xrightarrow{0}$	$E. \dashv$
$s_4 =$			
	E	$\xrightarrow{2}$	$T + .E$
	E	$\xrightarrow{4}$	$.T$
	E	$\xrightarrow{4}$	$.T + E$
	T	$\xrightarrow{4}$	$.F$
	T	$\xrightarrow{4}$	$.F * T$
	F	$\xrightarrow{4}$	$.IDENT$
$s_5 =$			
	F	$\xrightarrow{4}$	$IDENT.$
	T	$\xrightarrow{4}$	$F.$
	T	$\xrightarrow{4}$	$F. * T$
	E	$\xrightarrow{4}$	$T.$
	E	$\xrightarrow{4}$	$T. + E$
	E	$\xrightarrow{2}$	$T + E.$
	E	$\xrightarrow{0}$	$T + E.$
	S'	$\xrightarrow{0}$	$E. \dashv$
$s_6 =$			
	E	$\xrightarrow{4}$	$T + .E$
	E	$\xrightarrow{6}$	$.T$
	E	$\xrightarrow{6}$	$.T + E$
	T	$\xrightarrow{6}$	$.F$
	T	$\xrightarrow{6}$	$.F * T$
	F	$\xrightarrow{6}$	$.IDENT$
$s_7 =$			
	F	$\xrightarrow{6}$	$IDENT.$
	T	$\xrightarrow{6}$	$F.$
	T	$\xrightarrow{6}$	$F. * T$
	E	$\xrightarrow{6}$	$T.$
	E	$\xrightarrow{6}$	$T. + E$
	E	$\xrightarrow{4}$	$T + E.$
	E	$\xrightarrow{2}$	$T + E.$
	E	$\xrightarrow{0}$	$T + E.$
	S'	$\xrightarrow{0}$	$E. \dashv$
$s_8 =$			
	S'	$\xrightarrow{0}$	$E \dashv .$

Table 4: States processed while parsing $\mathbf{A} + \mathbf{B} + \mathbf{C} + \mathbf{D}$ under grammars EXP − L and EXP − R

$\mathbf{E} \longrightarrow \mathbf{T}$	$\mathbf{T} \longrightarrow \mathbf{F}$	$\mathbf{F} \longrightarrow \mathbf{IDENT}$
$\mathbf{E} \longrightarrow \mathbf{E} + \mathbf{E}$	$\mathbf{T} \longrightarrow \mathbf{T} * \mathbf{T}$	

Table 5: Grammar AMB

$S_0 =$		
$\mathbf{S'}$	$\xrightarrow{0}$	$.\mathbf{E} \dashv []$
$\mathbf{E}$	$\xrightarrow{0}$	$.\mathbf{T}[]$
$\mathbf{E}$	$\xrightarrow{0}$	$.\mathbf{E} + \mathbf{E}[]$
$\mathbf{T}$	$\xrightarrow{0}$	$.\mathbf{F}[]$
$\mathbf{T}$	$\xrightarrow{0}$	$.\mathbf{T} * \mathbf{T}[]$
$\mathbf{F}$	$\xrightarrow{0}$	$.\mathbf{IDENT}[]$
$S_1 =$		
$\mathbf{F}$	$\xrightarrow{0}$	$\mathbf{IDENT}.[a]$
$\mathbf{T}$	$\xrightarrow{0}$	$\mathbf{F}.[(Fa)]$
$\mathbf{E}$	$\xrightarrow{0}$	$\mathbf{T}.[(T(Fa))]$
$\mathbf{T}$	$\xrightarrow{0}$	$\mathbf{T}. * \mathbf{T}[(T(Fa))]$
$\mathbf{S'}$	$\xrightarrow{0}$	$\mathbf{E}. \dashv [(E(T(Fa)))]$
$\mathbf{E}$	$\xrightarrow{0}$	$\mathbf{E}. + \mathbf{E}[(E(T(Fa)))]$
$S_2 =$		
$\mathbf{E}$	$\xrightarrow{0}$	$\mathbf{E} + .\mathbf{E}[(E(T(Fa))), +]$
$\mathbf{E}$	$\xrightarrow{2}$	$.\mathbf{T}[]$
$\mathbf{E}$	$\xrightarrow{2}$	$.\mathbf{E} + \mathbf{E}[]$
$\mathbf{T}$	$\xrightarrow{2}$	$.\mathbf{F}[]$
$\mathbf{T}$	$\xrightarrow{2}$	$.\mathbf{T} * \mathbf{T}[]$
$\mathbf{F}$	$\xrightarrow{2}$	$.\mathbf{IDENT}[]$
$S_3 =$		
$\mathbf{F}$	$\xrightarrow{2}$	$\mathbf{IDENT}.[a]$
$\mathbf{T}$	$\xrightarrow{2}$	$\mathbf{F}.[(Fa)]$
$\mathbf{E}$	$\xrightarrow{2}$	$\mathbf{T}.[(T(Fa))]$
$\mathbf{T}$	$\xrightarrow{2}$	$\mathbf{T}. * \mathbf{T}[(T(Fa))]$
$\mathbf{E}$	$\xrightarrow{0}$	$\mathbf{E} + \mathbf{E}.[(E(T(Fa))), +, (E(T(Fa)))]$
$\mathbf{E}$	$\xrightarrow{2}$	$\mathbf{E}. + \mathbf{E}[(E(T(Fa)))]$
$\mathbf{S'}$	$\xrightarrow{0}$	$\mathbf{E}. \dashv [(+(E(T(Fa)))(E(T(Fa))))]$
$\mathbf{E}$	$\xrightarrow{0}$	$\mathbf{E}. + \mathbf{E}[(+(E(T(Fa)))(E(T(Fa))))]$
$S_4 =$		
$\mathbf{E}$	$\xrightarrow{2}$	$\mathbf{E} + .\mathbf{E}[(E(T(Fa))), +]$
$\mathbf{E}$	$\xrightarrow{0}$	$\mathbf{E} + .\mathbf{E}[(+(E(T(Fa)))(E(T(Fa)))), +]$
$\mathbf{E}$	$\xrightarrow{4}$	$.\mathbf{T}[]$
$\mathbf{E}$	$\xrightarrow{4}$	$.\mathbf{E} + \mathbf{E}[]$
$\mathbf{T}$	$\xrightarrow{4}$	$.\mathbf{F}[]$
$\mathbf{T}$	$\xrightarrow{4}$	$.\mathbf{T} * \mathbf{T}[]$
$\mathbf{F}$	$\xrightarrow{4}$	$.\mathbf{IDENT}[]$
$S_5 =$		
$\mathbf{F}$	$\xrightarrow{4}$	$\mathbf{IDENT}.[a]$
$\mathbf{T}$	$\xrightarrow{4}$	$\mathbf{F}.[(Fa)]$
$\mathbf{E}$	$\xrightarrow{4}$	$\mathbf{T}.[(T(Fa))]$
$\mathbf{T}$	$\xrightarrow{4}$	$\mathbf{T}. * \mathbf{T}[(T(Fa))]$
$\mathbf{E}$	$\xrightarrow{2}$	$\mathbf{E} + \mathbf{E}.[(E(T(Fa))), +, (E(T(Fa)))]$
$\mathbf{E}$	$\xrightarrow{0}$	$\mathbf{E} + \mathbf{E}.[(+(E(T(Fa)))(E(T(Fa)))), +, (E(T(Fa)))]$
$\mathbf{E}$	$\xrightarrow{4}$	$\mathbf{E}. + \mathbf{E}[(E(T(Fa)))]$
$\mathbf{E}$	$\xrightarrow{0}$	$\mathbf{E} + \mathbf{E}.[(E(T(Fa))), +, (+(E(T(Fa)))(E(T(Fa))))]$
$\mathbf{E}$	$\xrightarrow{2}$	$\mathbf{E}. + \mathbf{E}[(+(E(T(Fa)))(E(T(Fa))))]$
$\mathbf{S'}$	$\xrightarrow{0}$	$\mathbf{E}. \dashv [(+(+(E(T(Fa)))(E(T(Fa))))(E(T(Fa))))]$
$\mathbf{E}$	$\xrightarrow{0}$	$\mathbf{E}. + \mathbf{E}[(+(+(E(T(Fa)))(E(T(Fa))))(E(T(Fa))))]$
$\mathbf{S'}$	$\xrightarrow{0}$	$\mathbf{E}. \dashv [(+(E(T(Fa)))(+(E(T(Fa)))(E(T(Fa)))))]$
$\mathbf{E}$	$\xrightarrow{0}$	$\mathbf{E}. + \mathbf{E}[(+(E(T(Fa)))(+(E(T(Fa)))(E(T(Fa)))))]$
$S_6 =$		
$\mathbf{S'}$	$\xrightarrow{0}$	$\mathbf{E} \dashv .[(+(+(E(T(Fa)))(E(T(Fa))))(E(T(Fa)))), \dashv]$
$\mathbf{S'}$	$\xrightarrow{0}$	$\mathbf{E} \dashv .[(+(E(T(Fa)))(+(E(T(Fa)))(E(T(Fa))))), \dashv]$

Table 6: States produced while parsing $\mathbf{A} + \mathbf{B} + \mathbf{C}$ under grammar AMB