MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Artificial Intelligence
Memo. No. 184

MAC-M-395
March 1969

Parsing Key Word Grammars

William A. Martin

Key word grammars are defined to be the same as context free grammars, except that a production may specify a string of arbitrary symbols. These grammars define languages similar to those used in the programs CARPS[1] and ELIZA[2]. We show a method of implementing the LR(k) parsing algorithm for context free grammars which can be modified slightly in order to parse key word grammars. When this is done the algorithm can use many of the techniques used in the ELIZA parse. Therefore, the algorithm helps to show the relation between the classical parsers and key word parsers.

1. The LR(k) Parsing Scheme

We indicate the basic idea of the LR(k) parsing scheme by giving an example. A formal description and discussion of the method can be found in Knuth[3]. Consider the following context free grammar:

1 $\overline{S} \rightarrow S\#$

2 $S \rightarrow E + E$

3 $E \rightarrow F * F$

4 $E \rightarrow F$

5 $F \rightarrow x$

6 $F \rightarrow y$

Fig. 1

The string x * y + x# lies in the language generated by this grammar. We can parse this string with the LR(k) algorithm. This algorithm makes one pass through the string from left to right. The parameter k refers to the number of characters which the algorithm "looks ahead" at each step. We will take k = 1. The complete parse of the string is shown in Fig. 2.

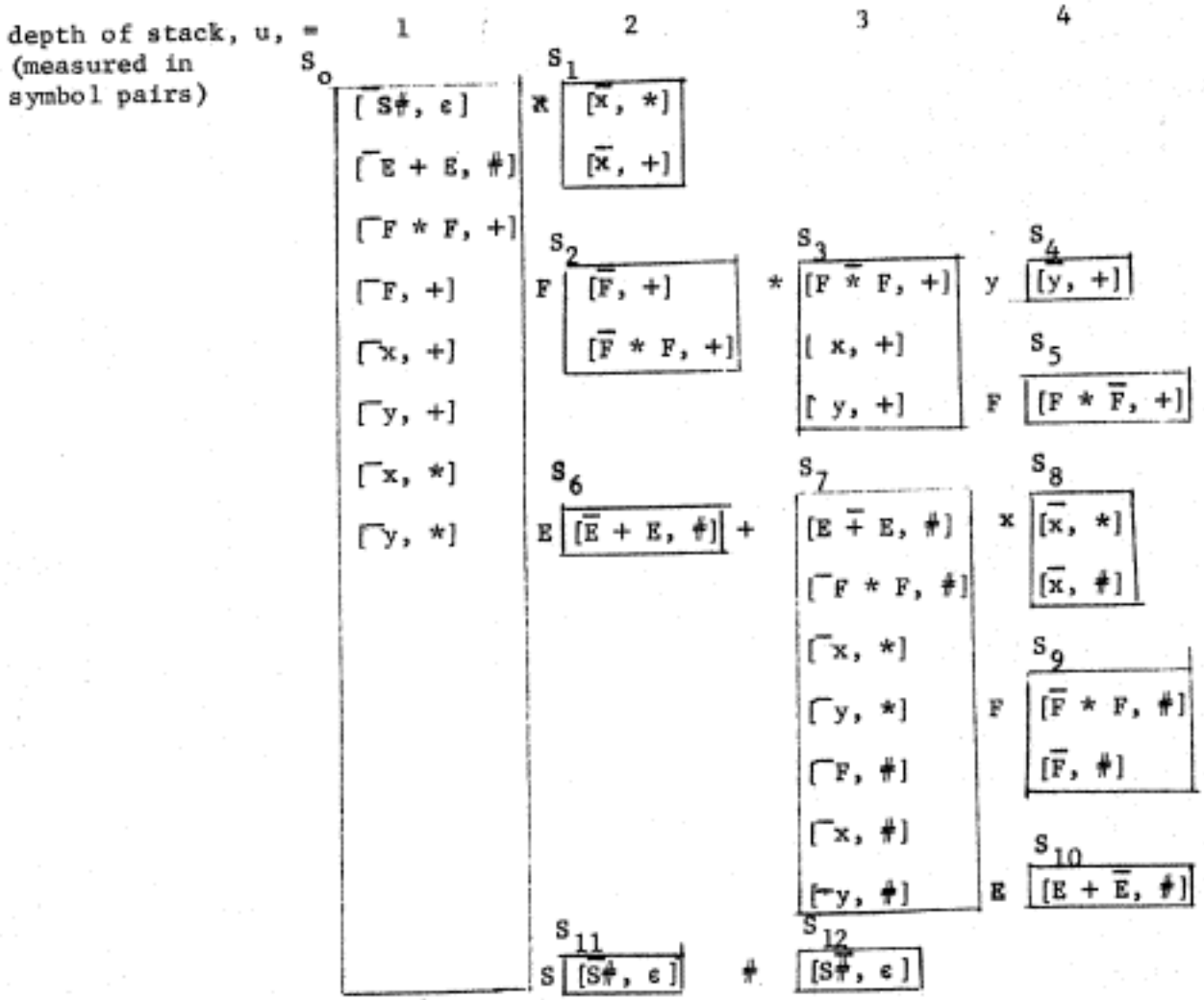


Fig. 2.

Parsing the string x * y + x#

This figure shows the successive stages of the push-down-stack used in the parse. Each rectangle is named by the symbol S_i at its top left; the successive stages of the stack are:

S_0

$S_0 \times S_1$

$S_0 F S_2$

$S_0 F S_2 * S_3$

$S_0FS_2 * S_3yS_4$

$S_0FS_2 * S_3FS_5$

S_0ES_6

$S_0ES_6 + S_7$

$S_0ES_6 + S_7 \times S_8$

$S_0ES_6 + S_7FS_9$

$S_0ES_6 + S_7ES_{10}$

S_0SS_{11}

$S_0SS_{11}\# S_{12}$

Let us see how the information in the successive rectangles, S_i, and corresponding stages of the stack are generated. Each S_i is a set of states of the form [right side of a production, terminal character] with a horizontal bar placed just before one character on the right side of the production. The terminal character is the character which must be the next input character if a reduction of the stack corresponding to the production whose right side is given in the state is to be made. To form S_0, we ask what productions could possibly lead to the first character of any input string. Since all derivations of an acceptable input string must start with production 1, $\overline{S} \rightarrow S\#$ we start S_0 with the state $[\overline{S}\#, \epsilon]$, indicating that we are looking for the string $S\#$ followed by no input character and indicating with the placement of the - before the S that we have not yet found any of the characters of this string. Now from the grammar we see that in order to find the first character of this string, S, which is to be followed by a # we must find the string E + E followed by a # so we add the state $[\overline{E} + E, \#]$ to S_0. Similarly, to find an E followed by a + we must find either F * F (corresponding to production 3)

followed by a + or the string F (corresponding to production 4) followed by a +. This process of adding to S_0 all the states which we should be looking for as a consequence of the states already in S_0 is called "computing the closure of S_0". The complete closure of S_0 is shown in Fig. 2.

Now that we have S_0 we place it on the stack. We examine each state in S_0 to see if we have found all of the specified characters in any of them. We have not, so we add the first input character, x, to the stack. We then compute S_1 by placing in S_1 every state in S_0 which has the $\bar{}$ immediately to the left of the character, x, which was just placed on the stack. We place the $\bar{}$ over the x to indicate that the x has been "found". S_1 thus has two states $[\bar{x}, *]$ and $[\bar{x}, +]$. Next we compute the closure of S_1. It is already closed. We then place S_1 on the stack. Now we proceed as we did when we placed S_0 on the stack. We look to see if any states in S_1 have all of the characters found, and both of them do. Since the next input character is * we ignore the second state, $[\bar{x}, +]$, and make the reduction, $x \rightarrow F$, corresponding to the first. x is said to be the current "handle". To make this reduction we remove x and S_1 from the stack and replace them with F. Then we form S_2 as the closure of those states in S_1 which have an F preceded by $\bar{}$. We continue as above until the parse is completed with the generation of S_{12}.

Note that there are only a finite number of possible states and so there are only a finite number of distinct S_i. It is possible to compute once and for all each S_i which will occur in parsing any string which is generated by a given grammar which can be parsed by this algorithm. Thus one could set up an array which would give the action which the parser should take for each combination of an S_i and with an input symbol.

The parse is then reduced to table look up and the mechanism is very similar to a precedence algorithm parse. However, if there are as many as 200 productions. This array could be very large (even if simplifications to remove redundant cases are made).

2. An Implementation of the LR(1) Parsing Scheme

We now introduce an alternative approach. The array approach summarizes each state S_i in a single number. However, if the next state is very "similar" to the last state then an acceptable alternative is to try to represent the state by many numbers, only a few of which will change with each change of state. The push-down-list is then used to save only those numbers which change. In implementing this approach we assume that the depth of the stack can always be specified as an entry of the array which defines the state. As we will see, the state can then be defined by alloting entries in this array for each combination of handle and character which can follow it in some parse. A number of entries equal to the number of characters in the handle would be alloted for each such combination. However, in an attempt to simplify the procedure without destroying its usefulness we will not keep this much information. We will only keep a list of the characters which cannot follow the right side of a given production in any sentential form, then each right side need only appear once in the array defining the state, instead of once for each character which can follow it in some sentential form. Then we can specify the array defining the state as containing one symbol for each symbol in each production of the grammar to be parsed. For the grammar above the array will have 17 entries.

array entry	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
corresponding symbol	S	#	$(\overline{S})$	E	+	E	(S)	F	*	F	(E)	F	$(E)^{(*)}$	x	(F)	y	(F)

Fig. 3

Entries 1 and 2 correspond to symbols 1 and 2 on the right side of production 1. Entry 3 corresponds to the symbol on the left side of production 1. The remaining entries correspond to the remaining productions in the same way. The entries corresponding to the left side of a production are filled with a pointer to a function which reduces the stack if that production is found as a "handle" (in the sense explained in section 1) not followed by any of the specified characters. For example, entry 13 above says that the reduction $F \rightarrow E$ should only be made if the next input character is not *. We call the above array the state array. We also set up a second array whose function is to describe the entries in the state array. This second array is called the property array and its entries are in one-to-one correspondence with the entries of the state array. If an entry in the state array corresponds to a terminal, the corresponding entry in the property array is zero. If an entry in the state array corresponds to a non-terminal on the right side of a production, the entry in the property array is a negative number whose absolute value is a pointer to a list of every entry in the state array which corresponds to the first character on the right side of a production whose left side is this non-terminal. (This list is used to form closures efficiently.) If an entry in the state array corresponds to a non-terminal which is the left side of a production, the corresponding entry in the property array is a pointer to a list of those entries in the state array which correspond to an appearance of this non-terminal on the right side of some production. These entries

in the property array and the entries in the state array corresponding to the left side of a production are made once and for all and do not change during a parse. The current state of the parse is kept in the entries of the state array corresponding to the symbols on the right sides of productions and on the push-down-stack.

Initially the push-down-stack is empty and these state array entries are all zero. We then change the state array to represent the state S_0 as follows. Referring to Fig. 2 we see that the stack has depth 1 ($u = 1$) after S_0 is placed on it. During the parse the stack grows deeper and is then reduced, but whenever S_0 is on the top of the stack, the stack has depth 1. Therefore, S_0 can be interpreted as specifying that we are looking for the character S in production 1, E in production 2, F in productions 3 and 4, x in production 5 and y in production 6, when the stack is of depth 1. We thus set the state array to indicate this by placing a 1 in the entries corresponding to these characters. The array then has the form A_0 show in line 3 of Fig. 4. The stack is empty.

We now describe the procedure for going from state A_0 to state A_1, which corresponds to the procedure for going from S_0 to S_1 in Fig. 2. Each input character has associated with it a list of the entries in the state array which correspond to that character. The current input character here, x, thus has entry 14 associated with it. We then go to entry 14 and see if it contains a 1, indicating that x is wanted at the current push-down-stack level, which is 1. It is and so we advance to the next entry, 15. Checking the corresponding entry in the property array we see that we should make the reduction $x \rightarrow F$. Since the handle is only one character long the push-down-stack

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
	S	#	($\bar{S}$)	E	+	E	(S)	F	*	F	(E)	F	(E)(*)	x	(F)	y	(F)	depth, u
A_0:	1	0		1	0	0		1	0	0		1		1		1		1
stack: ()																		
sentential form: x * y + x#																		
input: x																		
A_2:	1	0		1	0	0		1	2	0		1		1		1		2
stack: ((0 9))																		
sentential form: F * y + x#																		
input: *																		
A_3:	1	0		1	0	0		1	2	3		1		3		3		3
stack: (|(0 9)|(0 10)(1 14)(1 16)|)																		
sentential form: F * y + x#																		
input: y																		
A_6:	1	0		1	2	0		1	0	0		1		1		1		2
stack: (|(0 5)|																		
sentential form: E + x#																		
input: +																		
A_7	1	0		1	2	3		3	0	0		3		3		3		3
stack: (|(0 5)|(0 6)(1 8)(1 12)(1 14)(1 16)|)																		
sentential form: E + x#																		
input: x																		
A_{11}	1	2		1	0	0		1	0	0		1		1		1		2
stack: (|(0 2)|)																		
sentential form: S#																		
input: #																		

Fig. 4

Entries in the state array at the steps during the parse when a new input character is examined.

depth remains 1. The property list entry tells us that entries 8, 10, and 12 correspond to F. Checking entry 12 we see that it is 1 so we advance to entry 13. Here we see that a reduction should be made if the next input character is not *, but it is *, so we abandon this and check entry 10. Entry 10 is 0 indicating that that this F is not needed. Finally we check entry 8. Entry 8 is 1 so we advance to entry 9. We see from the property list that entry 9 corresponds to a terminal, *, so we place a 2 in entry 9 indicating that we need a * at depth 2. We save the old value of entry 9 on the push-down-stack, to be restored when the stack is reduced back to depth 1. This brings us to state A_2 in Fig. 4, which corresponds to S_2 in Fig. 2. The next input character is a * which sends us to entry 9, entry 9 contains a 2 and so we advance to entry 10. The corresponding entry in the property array tells us that entry 10 corresponds to a non-terminal, so we place a 3 in entry 10, saving the previous contents on the push down list, we then obtain the list of entries needed to compute the closure. These are 14 and 16; we go to 14 and 16 and place a 3 in them, again saving the old contents. Since 14 and 16 correspond to terminals the closure is complete. This brings us to state A_3 in Fig. 4. Proceeding in the way we parse the input string as shown in Fig. 4.

3. Key Word Grammars

We define a key word grammar to be the same as a context free grammar except that the set of terminal characters is left unspecified and productions may contain arbitrary strings of the unspecified terminal characters. Since we can't list all of the terminal characters we let α stand for any string of zero or more terminal characters. A key word grammar is a set of productions of the form $A_p \rightarrow X_1 \ldots X_{n_p}$ where each X_i is either an intermediate, a

terminal, or the symbol α, and A_p is an intermediate. The strings generated by the grammar are thus patterns containing the symbol α. A string lies in the language generated by the grammar if it can be made to match one of the patterns generated by the grammar.

4. Parsing Key Word Grammars

Obviously, key word grammars are as general as context free grammars and so there will be keyword grammars which cannot be parsed with an algorithm less powerful (in some sense) than a non-deterministic push-down automaton. At the other end of the scale, since the strings, α, may contain any terminal characters the precedence relation = holds between every pair of strings of terminal characters and so a precedence algorithm may not be sufficient to parse keyword grammars.

We give here a variation of the LR(1) algorithm which seems to have enough power to parse interesting keyword grammars. If the algorithm is too slow in practice one might investigate a precedence algorithm. If it is not powerful enough we could expand it to LR(k).

The algorithm will not parse all keyword grammars. We therefore should define a test which a grammar must pass which will guarantee that the grammar can be parsed by the algorithm. One of the restrictions we make is that no production can have two adjacent α's or an α as the rightmost character of the right side of a rule.

5. A Key Word Grammar Parsing Algorithm

Our algorithm is a modification of the procedure given in Section 2. There, we looked for the characters of a production one by one. Now consider

the production A → a α b. The α means that after we find the "a" we should begin looking for a "b" and there can be any number of intervening characters. Thus, in the notation in Section 2, if we start looking for "b" at depth 2 we want to find "b" at depth 2 or any greater depth. We will indicate this by placing a - 2 instead of a 2 in the state array entry. Now consider the grammar

1. A → a α B
2. A → a b B
3. B → c

This grammar is ambiguous because the string a b c has the two parsings:

Yet we do not want to throw this grammar out. We can use it to express the useful idea that any string starting with "a" and ending with "b" should be matched with production 1 unless it has the specific form abc, in which case production 2 should be used. Let us see what this implies for the modification of the procedure in Section 2. If we try to parse the string "abc" we will find the "a" at depth 1, so we begin looking for a "c" at any depth ≥ 2 and a "b" at depth 2. We find a "b" at depth 2 and so we look for the "c" at depth 3. But we are already looking for the "c" at any depth ≥ 2. Therefore, we let the specific depth 3 dominate the "≥ 2" specification. We place the latter on a temporary list to be restored if we go on to depth 4, since we don't want to reject a string like abdc. Finally, in a conflict between "≥ 2" and "≥ 4", we would allow "≥ 4" to dominate, pushing "≥ 2" onto the main stack.

This is the idea of our modification to Section 2. In order to implement it we indicate in the property array whether or not each character on the right side of some production is preceded by α, but we do not put any entries in the state array for the α's. Then, during parsing, we enter a negative number instead of a positive number in the state array if the entry corresponds to a character preceded by α or if we are computing the closure of such an entry. Note that on making reductions handles corresponding to the same production can be of different lengths depending on the length of the strings matched by the α's. Therefore we must use the entries in the state array to find the left end of a handle.

6. An Example

Consider the grammar:

1 $\overline{S} \rightarrow S\#$

2 $S \rightarrow \alpha E + E$

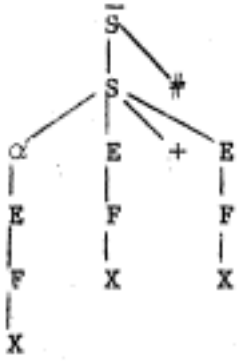
3 $E \rightarrow F*F$

4 $E \rightarrow F$

5 $F \rightarrow x$

6 $F \rightarrow y$

Given the string XX + X# the algorithm finds the parsing



Major steps in the parse are shown in Fig. 5. Note that if the reduction of the initial X to E prevented a correct reduction the string would be rejected. There are no grammars for which the algorithm will accept incorrect strings, but there are some for which it will reject correct ones.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
	S	#	($\bar{S}$)	E	+	E	(S)	F	*	F	(E)	F	(E)	X	(?)	y	(F)	level
A_0:	1	0		-1	0	0		-1	0	0		-1		-1		-1		1
stack:	()																	
sentential form:	XX + X#																	
A_1:	1	0		-1	2	0		-1	0	0		-1		-1		-1		2
stack:	(|(0 5)|)																	
sentential form:	EX + X#																	
A_2:	1	0		-1	2	0		-1	0	0		-1		-1		-1		2
stack:	(|(0 5)|)																	
sentential form:	EE + X#																	
A_3:	1	0		-1	2	3		-1	0	0		-1		-1		-1		3
stack:	(|(0 5)|(0 6)|)																	
sentential form:	EE + X#																	
A_4:	1	2		-1	0	0		-1	0	0		-1		-1		-1		2
stack:	(|(0 2)|)																	
sentential form:	S#																	

Fig. 5

Parsing the string XX + X with the new algorithm.

7. Conclusion

Exploring this idea further would make an interesting project for someone interested in parsing. For example, we could use all the ideas Weizenbaum

uses such as precedence among productions when one calls for a stack reduction and the other doesn't. Also, we can parse any key word grammar by modifying Early's[4] procedure along similar lines.

8. References

1. E. Charniak, "CARPS, a Program Which Solves Calculus Word Problems", MAC-TR-51, MIT, 1968.

2. J. Weizenbaum, "ELIZA"--A Computer Program for the Study of Natural Language Communication Between Man and Machine", CACM, Vol. 9, No. 1, Jan. 1966.

3. D.E. Knuth, "On the Translation of Languages from Left to Right", Information and Control 8, 1965.

4. J. Early, "An N^2-Recognizer for Context Free Grammars", Dept. of Computer Science Report, Carnegie-Mellon Univ., Pittsburgh, Pennsylvania, Sept. 1967.