ON FORMALIZATIONS OF MARCUS' PARSER

R. Nozohoor-Farshi
Dept. of Computing Science, University of Alberta, Edmonton,
Canada T6G 2H1

Abstract: LR(k,t), BCP(m,n) and LRRL(k) grammars, and their relations to Marcus parsing are discussed.

1. Introduction

In his IJCAI-83 paper [1], R.C. Berwick suggested that the stripped down version of Marcus' parser [4] (i.e., with no features or transformations) can be formally characterized by LR(k,t) parsers [7,8]. Berwick's COLING-84 paper [2] seems to suggest that Marcus-style parsers may be adequately formalized by bounded context parsable BCP(m,n) grammars[8,9].

In this paper we show that both classes of LR(k,t) and BCP(m,n) grammars are inadequate means to formalize Marcus' mechanism even when it is applied to parsing bare context-free grammars. We briefly describe a new class of unambiguous context-free grammars, LRRL(k), for which deterministic non-canonical bottom-up table driven parsers are generated automatically. These parsers employ k-symbol fully reduced right context in making parsing decisions. LRRL(k) grammars include as a subset those context-free grammars that are parsable by Marcus' partially top-down method.

2. Operation of Marcus' parser

Let us first recall that Marcus' parser has two data structures: a pushdown stack which holds the constructs yet to be completed, and a finite size buffer which holds the lookahead symbols. The lookaheads can be completed constructs as well as bare terminals. In addition, the parser has three basic operations:

(1) **Attach**: attaches a constituent in the buffer to the current active node (stack top).

(2) **Create (push)**: creates a new active node, i.e., when the parser decides that the first constituent(s) in the buffer begin a new higher constituent, a new node of the specified type is created and pushed on the stack. However the create operation has a second mode in which the newly created node is first attached to the old active node, and then pushed on the stack. Marcus indicates this by use of "attach a new node of 'type' to active node" in the grammar rules. Following Ritchie [6], we use a shorter notation: 'cattach' for this second mode.

(3) **Drop (pop)**: pops the top node of the stack (CAN). However if this node is not attached to a higher level node, it will be dropped in the first position of the window defined on the buffer. Marcus uses different notations, namely "drop" and "drop into buffer", in the grammar to indicate the effect of drop operations. This suggests that a grammar writer must be aware of the attachment of the current active node. Here, we adhere to his provision about differentiating between these two modes of drop operations. However we feel that there is no need for such a provision since PARSIFAL (the grammar interpreter) can take care of that by inserting an unattached node into the buffer, and the grammar can test the contents of the buffer to see if such insertion has taken place.

The three basic operations plus "attention shift" and "restore buffer" (forward and backward window movements on the buffer) are sufficient for parsing some context-free grammars which we informally denote by MP(k) (i.e., Marcus parsable with k lookaheads).

Now let us consider the context-free grammar G_1:

(1) S'→S
(2) S →d
(3) S →A S B
(4) A→a
(5) A→a S
(6) B→b

The following gives a Marcus-style parser for $L(G_1)$, i.e., a grammar G_2 written in a PIDGIN-like language that can be interpreted by PARSIFAL. The symbols inside square brackets refer to the contents of buffer positions, except [CAN=] which indicates the current active node. The grammar has no attention shift rules.

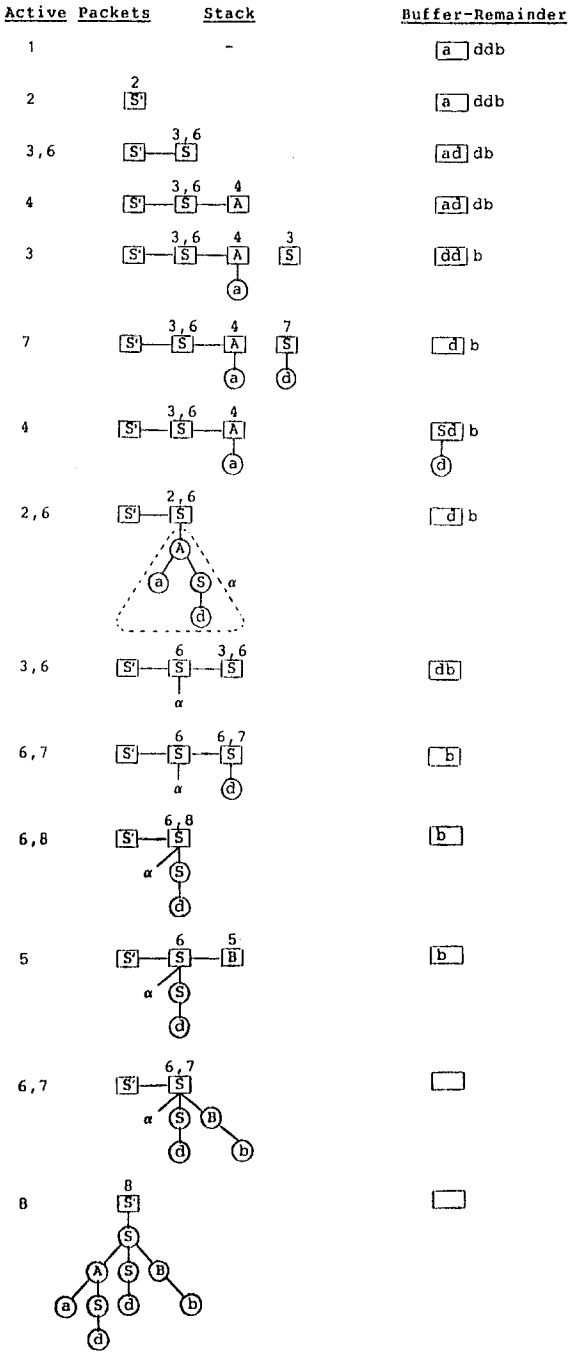
G_2:

Packet 1 : Initial rule.
[a or d] create S'; activate 2.
Packet 2 : Create and attach an S node.
[true] deactivate 2; cattach S; activate 3 and 6.
Packet 3 : S-parsing.
[d] attach first; deactivate 3; activate 7.
[a] cattach A; activate 4.
[Sb] attach first; deactivate 3; cattach B; activate 5.
Packet 4 : A-parsing.
[a] attach first; create S; activate 3.
[Sb] drop CAN.
[Sa or Sd] attach first; drop CAN; deactivate 3; activate 2.
Packet 5 : B-parsing.
[b] attach first; drop CAN; activate 7.
Packet 6 : Completion of an attached S node.
[true] drop CAN; activate 8.
(with priority p_1< default priority)
Packet 7 : Completion of an unattached S node.
[true] drop CAN into buffer. (with priority p_2<p_1)
Packet 8 : B-prediction.
[CAN=S] [b] deactivate 8; cattach B; activate 5.
[CAN=S'] [empty] "Parse is finished".

In the Marcus parser active packets are associated with the active node, that is, when a new node is created, some packets will usually be activated as well. Unless a packet is deactivated explicitly this association remains with the node. So when a node on the stack becomes the active node again as a result of 'pop' operations, its associated packets will be reactivated.

We do not attempt to show formally the equivalence of G_1 and G_2, since there is no formal characterization of Marcus-style parsers yet. However one may, by going through examples, convince oneself that the parser given in PIDGIN parses $L(G_1)$. Such an example is illustrated in detail next.

Example : The following diagrams illustrate the parsing of the sentence $addb \in L(G_1)$ by the parser described by G_2. The symbols inside the boxes are on the stack, and those inside the circles are already attached to a higher level symbol on the stack. The numbers shown above each stack node are the packet numbers associated with that node. G_2 uses a buffer of size 2 shown on the right.



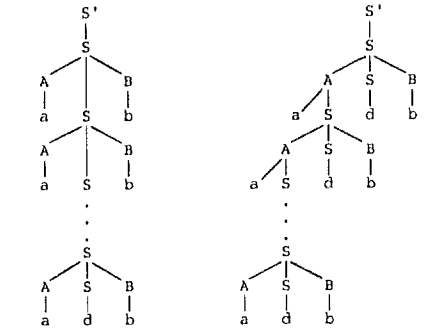
This example shows the power of the Marcus parser in employing completed subtrees as lookaheads. The grammar $\mathbf{G}_1$ is not LR(k) for any fixed k. Any *a* can be reduced to an *A* via production $A \rightarrow a$ or can be considered as a first symbol in production $A \rightarrow aS$ (i.e., a reduce/shift conflict in LR parser). However, in the first case *a* is followed by an *Sb*, and in the latter by an *Sd* or an *Sa*. By postponing the parsing decisions about the completion of *A*'s, Marcus' parser is able to produce the correct parse.

3. LR(k,t) Grammars

LR(k,t) grammars were originally proposed by Knuth in his landmark paper on parsing of LR(k) grammars [3], and later developed by Szymanski [7,8]. Essentially the LR(k,t) parsing technique is a non-canonical extension of the LR(k) technique, in which instead of the reduction of the handle (the leftmost phrase) of a right sentential form, we must be able to determine that in any sentential form at least one of the t (a fixed number) leftmost phrases is reducible to a specific non-terminal. In other words, a grammar G is not LR(k,t) if in parsing of an input sentence the decision about reduction of t or more questionable phrases in a sentential form needs to be delayed. The reduction decision is reached by examining the whole left context and k symbols to the right of a phrase in a sentential form.

Now, it is easy to see that $\mathbf{G}_1$ is not LR(k,t) for any finite numbers k and t. For given k and t, $\mathbf{L}(\mathbf{G}_1)$ includes sentences with prefix a^n where $n>k+t$. In such sentences t initial *a*'s have different interpretations depending on the other parts of the sentences. For example consider the two sentences:

(I) $a^n db^n$ $\quad n>k+t$ $\qquad$ (II) $a^n (db)^n$ $\quad n>k+t$



In (I) all t initial *a*'s must be reduced to *A*'s, while in (II) none of them is a phrase. Therefore an LR(k,t) parser will need to delay reduction of more than t possible phrases in parsing of a sentence with a prefix a^n, $n>k+t$, and thus $\mathbf{G}_1$ is not LR(k,t) for any given k and t. In fact, LR(k,t) parsers put a limit t on the number of delayed decisions at any time during the parsing. In Marcus parsing, depending on characteristics of the grammar there may be no limit on this number.

We have shown that $MP(k) \not\subset LR(k',t)$ for any k' and t. An interesting question is whether $LR(k,t) \subset MP(k')$ for some k'. The answer is negative. Consider the $LR(0)=LR(0,1)$ grammar $\mathbf{G}_2$:

$S \rightarrow A$	$A \rightarrow cA$	$A \rightarrow a$
$S \rightarrow B$	$B \rightarrow cB$	$B \rightarrow b$

With any finite buffer, Marcus' parser will be flooded with *c*'s, before it can decide to put an *A* node or a *B* node on the stack. The weakness of Marcus' parser is in its insistence on being partially predictive or top-down. Purely bottom-up LRRL(k) parsers remedy this shortcoming.

4. BCP(m,n) Grammars

The bounded context parsable grammars were introduced by Williams [9]. In parsing these grammars we need to be able to reduce at least one phrase in every sentential form (in a bottom-up fashion) by looking at m symbols to the left and n symbols to the right of a phrase. BCP-parsers use two stacks to work in this fashion.

It is trivial to show that BCP grammars are unsuitable for formalizing the Marcus parser. A BCP-parser ignores the information extractable from the left context (except the last m symbols). Whereas in the Marcus parser, the use of that information is the compelling reason for deployment of the

packeting mechanism. In fact there are numerous simple LR(k) grammars that are not BCP, but are parsed by the Marcus parser. An example is the grammar G_4:

S →aA	S→bB
A→d	A→cA
B→d	B→cB

A Marcus-style parser after attaching the first symbol in an input sentence will activate different packets to parse *A* or *B* depending on whether the first symbol was *a* or *b*. However, a BCP-parser cannot reduce the only phrase, i.e., *d* in the sentences *ac...cd* and *bc...cd*. Because a number of *c*'s more than m shields the necessary context for reduction of *A→d* or *B→d*.

5. LRRL(k) Grammars

LRRL(k) parsing basically is a non-canonical bottom-up parsing technique which is influenced by the "wait and see" policy of Marcus' parser. By LRRL(k) grammars, we denote a family of grammar classes that are parsed *l*eft to *r*ight with *k* *r*educed *l*ookaheads in a deterministic manner. The difference between these classes lies in the nature of lookaheads that they employ. Roughly, the class with more 'complex' lookaheads includes the class with 'simpler' lookaheads. Here, we discuss the basic LRRL(k) grammars. Further details about LRRL(k) grammars and the algorithm for generation of basic LRRL parsers are given in [5].

A basic LRRL(k) parser employs k-symbol fully reduced right contexts or lookaheads. The k fully reduced right context of a phrase in a parse tree consists of the k non-null deriving nodes that follow the phrase in the leftmost derivation of the tree. Thus these nodes dominate any sequence of k subtrees to the immediate right of the phrase that have non-null frontiers. This generalized lookahead policy implies that when a questionable handle in a right sentential form is reached, the decision to reduce it or not may be reached by parsing ahead a segment of the input that can be reduced to a relevant fully reduced right context of length k. For example, in parsing a sentence in $L(G_1)$, after seeing the initial *a* there is a shift-reduce conflict as to whether we should reduce according to rule (4) or continue with the rule (5). However the 2-symbol fully reduced context for reduction is *SB*, and for the shift operation is *SS*, which indicates a possible resolution of conflict if we can parse the lookaheads. Therefore we postpone the reduction of this questionable phrase and add two new auxiliary productions *SUBGOAL-RED(4)→SB* and *SUBGOAL-SHIFT→SS*, and continue with the parsing of these new constructs. Upon completion of one of these productions we will be able to resolve the conflicting situation. Furthermore, we may apply the same policy to the parsing of lookahead contexts themselves. This feature of LRRL(k) parsing, i.e., the recursive application of the method to the lookahead information, is the one that differentiates this method from any other. The method is recursively applied whenever the need arises, i.e., at ambivalent points during parsing.

Note that the lookahead scheme does not allow us to examine any remaining segment of the input that is not a part of the lookahead context. The parsed context is put in a buffer of size k, and no reexamination of the segment of the input sentence that has been reduced to this right context is carried out. In addition, the right context which is k symbols or less does not contain a complete phrase, i.e., the symbols in the right context do not participate in any future reductions involving only these symbols.

The parsing algorithm for an LRRL(k) grammar is based on construction of a Characteristic Finite State Machine. A CFSM is rather similar to the deterministic finite automaton that is used in LR(k) parsing for recognition of viable prefixes. However there are three major differences:

(1) The nature of lookaheads. The lookaheads are fully reduced symbols as opposed to bare terminals in LR(k) parsers.
(2) Introduction of auxiliary productions.
(3) Partitioning of states which conceals conflicting items.

The information extracted from this machine is in tabulated form that acts as the finite control for the parsing algorithm.

The basic LRRL grammars, when augmented with attributes or features, generate a class of languages that includes the subsets of English which are parsable by a Marcus type parser. Thus introduction of LRRL grammars provides us with a capability for automatic generation of Marcus style parsers from a context-free base grammar plus the information about the feature set, their propagation and matching rules, and a limited number of transformational rules (e.g., auxiliary inversion and handling of traces). We believe that such a description of a language in a declarative grammar form is much more understandable than the procedurally defined form in Marcus' parser. Not only does the presence of parsing notions such as create, drop, etc. in a PIDGIN grammar make it difficult to determine exactly what language (i.e. what subset of English) is parsed by the grammar, but it is also very hard to determine whether a given arbitrary language can be parsed in this style and if so, how to construct a parser. Furthermore, modification of existing parsers and verification of their correctness and completeness seems to be unmanageable.

We may pause here to observe that LRRL parsing is a bottom-up method, while Marcus' parser is not strictly a bottom-up one. In fact it proceeds in a top-down manner and when need arises it continues in a bottom-up fashion. However, as Berwick [1] notes, the use of top-down prediction in such a parser does not affect its basic bottom-up completion of constructs. In fact the inclusion of MP(k) grammars in the more general class of LRRL(k) grammars is analogous to the inclusion of LL(k) grammars in the class of LR(k) grammars. In the Marcus parser incomplete nodes are put on the stack, while in a bottom-up parser completed nodes that seek a parent reside on the stack.

6. Conclusion

We have shown that the class of context-free grammars parsable by a Marcus-type parser is neither a subclass of LR(k,t) nor a subclass of BCP(m,n) grammars. We have introduced LRRL(k) grammars, which formalize the concepts of Marcus parsing in a purely bottom-up parser. One may consider the lookahead policy used in basic LRRL(k) grammars as the opposite extreme to the one employed in LR(k) grammars. In LR(k) parsing the lowest level of nodes, i.e., terminals are used as lookaheads, while in basic LRRL(k) parsing the highest level nodes that follow the current construct act as lookaheads. A modified version of these grammars combines the two policies. The most general class of LRRL(k) grammars which is defined in [5] considers lookaheads at arbitrary levels. It can be shown that for a fixed k, this class of grammars is the largest known class that generalizes the concepts of LR(k) parsing while retaining the property that membership of an arbitrary grammar in the class is still decidable.

Acknowledgements

The author is indebted to Dr. L.K. Schubert for his suggestions and careful review of the draft of this paper. The research was supported by the Natural Sciences and Engineering Research Council of Canada Operating Grant A8818 under Dr. Schubert's supervision.

References

[1] R.C. Berwick. A deterministic parser with broader coverage. IJCAI 83, Proceedings of the 8th International Joint Conference on Artificial Intelligence, pp. 710-712. August 1983.
[2] R.C. Berwick. Bounded context parsing and easy learnability. COLING 84, Proceedings of the 10th International Conference on Computational Linguistics, pp. 20-23. Stanford University. July 1984.
[3] D.E. Knuth. On the translation of languages from left to right. Information and Control, vol. 8, pp. 607-639. 1965.
[4] M.P. Marcus. A Theory of Syntactic Recognition for Natural Langauge. MIT Press, Cambridge, MA. 1980.
[5] R. Nozohoor-Farshi. LRRL(k) grammars: a left to right parsing technique with reduced lookaheads. Ph.D. thesis in preparation, Dept. of Computing Science, University of Alberta. 1986.
[6] G.D. Ritchie. The implementation of a PIDGIN interpreter. Automatic Natural Language Parsing, eds. K. Spark Jones and Y. Wilks, pp. 69-80. Ellis Horwood, Chichester, England. 1983.
[7] T.G. Szymanski. Generalized bottom-up parsing. Ph.D. thesis, Dept. of Computer Science, Cornell University. 1973.
[8] T.G. Szymanski and J.H. Williams. Non-canonical extensions of bottom-up parsing techniques. SIAM Journal of Computing, vol. 5, no. 2, pp. 231-250. June 1976.
[9] J.H. Williams. Bounded context parsable grammars. Information and Control, vol. 28, pp. 314-334. 1975.