BUFFERED SHIFT-REDUCE PARSING*

Bing Swen (Sun Bin)
Dept. of Computer Science, Peking University, Beijing 100871, China
bswen@cs.pku.edu.cn

Abstract *A parsing method called buffered shift-reduce parsing is presented, which adds an intermediate buffer (queue) to the usual LR parser. The buffer's usage is analogous to that of the wait-and-see parsing, but it has unlimited buffer length, and may serve as a separate reduction* (*pruning*) *stack. The general structure of its parser and some features of its grammars and parsing tables are discussed.*

1. Introduction

It is well known that the LR parsing is hitherto the most general shift-reduce method of bottom-up parsing [1], and is among the most widely used. For example, almost all compilers of mainstream programming languages employ the LR-like parsing (via an LALR(1) compiler generator such as YACC or GNU Bison [2]), and many NLP systems use a generalized version of LR parsing (GLR, also called the Tomita algorithm [3]).

In some earlier research work on LR(*k*) extensions for NLP [4] and generalization of compiler generator [5], an idea naturally occurred that one can make a modification to the LR parsing so that after each reduction, the resulting symbol may not be necessarily pushed onto the analysis stack, but instead put back into the input, waiting for decisions in the following steps. This strategy postpones the time of some reduction actions in traditional LR parser, partly deviating from leftmost pruning (Leftmost Reduction). This may cause performance loss for some grammars, with the gained opportunities of earlier error detecting and avoidance of false reductions, as well as later handling of ambiguities, which is significant for the processing of highly ambiguous input strings like natural language sentences. This also partly emulates the "Wait-and-See" strategy (with a 3-unit buffer) in deterministic parsing (Determinism Hypothesis)[6,7], but is not limited by predefined number of lookahead symbols.

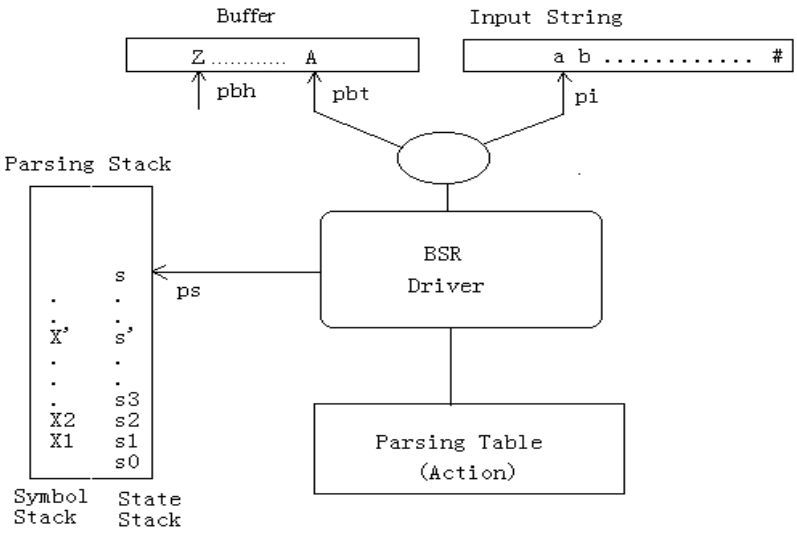
Based on these work and improvements, the author presented a new parsing method [8] called Buffered Shift-Reduce parsing (or BSR parsing for short), which systematically delays reduction actions by employing an intermediate buffer queue, providing the maximum opportunity of ambiguity lookup, whereas the previous reduce-and-putback approach provides only one production scope of lookup. In this paper we present the basic idea of this technique, describe the general structure of its parser, and discuss some features of its grammars and parsing tables.

2. Shift-reduce parsing with an intermediate buffer

To enable more control of the reduction nodes within the parse, we add a symbol buffer to the traditional shift-reduce parser, as shown in the following figure, where # stands for the end of input string, and *pi*, *pbh*/*pbt* and *ps* for pointers to input, buffer and parsing stack respectively. The parsing stack here includes both states and syntactical symbols. In practical parses, syntactical symbols are not necessarily maintained in the stack, for they can be deduced quickly from the top state of the state stack together with the production used in the reduction.

The basic flow of input tokens is that instead of being directly moved onto the symbol stack, they are first shifted into the buffer queue, where they may be reduced to other symbols and may also result other sequential reductions in the buffer and/or on the parsing stack. Symbols contained in the buffer are not limited to a fixed number, and will be finally shifted onto the symbol stack, where they are to be reduced to other symbols. Since the parser can look ahead any number of symbols before making shift/reduce decisions, and make reductions at tow separate nodes in the parse tree, it gets more control to handle conflicts and ambiguities.

* This work is supported by National Natural Science Foundation of China, project No. 69973005.



3. Actions and parsing algorithm

The general form of a BSR parser's configuration would be determined by 4 items, that is, a quadruple [*s*, *Z*, *A*, *a*], as shown in the above figure. And correspondingly, a production would be divided into more specific cases of the form $A \rightarrow \alpha \cdot X\ \beta\ Y * Z\ \gamma$, where '$\cdot$' and '$*$' correspond to boundaries of buffer and input respectively. Here we will only discuss triple configurations [*s*, *A*, *a*], where the parser can make shift-reduce decisions in the buffer using the tail symbol at a given stack state. The parser has the following categories of actions:

Buffer-shift: *Action*[*s*, *A*, *a*] = { **ps*++ = **pi*++; **ps*++ = *s*'; }
Stack-shift: *Action*[*s*, *A*, *a*] = { **p_symbol_stack*++ = **pbh*++; **ps*++ = *s*'; } // *pbh*!=*pbt*
Stack-reduce: *Action*[*s*, *A*, *a*] = { symbol stack top replaced by $A \rightarrow \beta$, **ps*++ = *s*'; }
Buffer-reduce: *Action*[*s*, *A*, *a*] = { buffer tail replaced by $A \rightarrow \beta$, **ps*++ = *s*'; }
Accept: *Action*[*s*, *A*, *a*] = { **ps*++ = *ACCEPT*; output "accept"; }
Error: *Action*[*s*, *A*, *a*] = { **ps*++ = *ERROR*; output "error"; }

With the above discussion, we establish the following algorithm for the buffered shift-reduce parsing (which is a linear algorithm).

***Input*:** an input token string *w* and a buffered shift-reduce parsing table *Action*[] of grammar *G*;
***Output*:** if *w* belongs to language *L*(*G*), then output a BSR parse of *w*; otherwise an error indication.
***Steps*:**

Initialization: push s_0 to the bottom of state stack, read *w* into the input buffer, set *ps* to the top of state stack, buffer queue to be empty *pbh=pbt*, and *pi* to the beginning of imput;

main_loop:
repeat forever begin
s := **ps*++; /* current state */
a := **pi*++; /* points to the next input token */
if *pbh* = *pbt* **then begin**
A := 0;
else begin
A := **pbt*; *Z* := **pbh*;
end
if *Action*[*s*, *A*, *a*] = (buffer-shift, *s*') **then begin**
**pbt*++ := *a*; **ps* := *s*';
end
if *Action*[*s*, *A*, *a*] = (stack-shift, *s*') **then begin**
/* shift *Z* onto the top of symbol stack – actually no need */
/* *Assert*(*pbh* <= *pbt*); */
pbh++; **ps* := *s*';
end

```
        else if Action[s, A, a] = (stack-reduce A→β , s')    then begin
                /* pop |β| symbols from top of the symbol stack, and push A – actually no need */
                /* pop |β| states from top of the state stack, and push s' onto: */
                ps -= |β|;    *ps := s';
        end
        else if Action[s, A, a] = (buffer-reduce A→β , s')    then begin
                /* pop |β| symbols from top of the symbol stack, and push A: */
                pbt -= |β|;    /* Assert(pbh <= pbt); */
                *pbt++ := A;    *ps := s';
        end
        else if Action[s, A, a] = ACCEPT and a= '#' then
                return SUCCESS;
        else    return ERROR;
end
```

As pointed above, this method may postpone some reduction actions as in LR parsing, with more shift actions being applied. This may help avoid some false reductions. Depending on the actual construction of action tables, the scope to deviate the leftmost pruning in buffered shift-reduce parsing may vary in a wide range.

4. BSR grammars and simplified parsing tables

Similarly to LR parsing, there would be 4 different ways to construct a buffered shift-reduce parsing table for a given grammar *G*, which correspond to 4 buffered shift-reduce parsers of different acceptance capability. The grammars accepted by all these parsers are called buffered LR grammar, or BLR grammar for short. The 4 categories of BLR are BLR(0), SBLR, BLR(*k*) and LABLR(*k*) respectively. The buffered shift-reduce parsing may also be applied to some precedence parsing methods. For example, we can construct an operator-precedence parser (with weak OP grammars) based on the buffered shift-reduce parsing. Those will, however, not be discussed here.

When constructing a BSR parsing table, both the input symbols and buffered symbols are significant for parsing decisions. Since we consider only one tail buffer symbol, we can further simply the construction of parsing tables by noticing that for this simplified parsing, the lookahead symbol may be nonterminals besides terminals. The dotted productions of corresponding parsing tables also take the usual form $A \to \alpha \cdot X \beta$, and parser configuration the binary form [*s*, *X*], where *X* may represent *A* (when buffer is not empty) or *a* (when buffer empty). In this regarding, we redefine the *First* and *Follow* sets in LR parsing as the following.

***First* set:**

Given any string of grammar symbols $\alpha \in (V_T \cup V_N)^*$, $First(\alpha)$ is defined as

$$First(\alpha) = \left\{ V \middle| \alpha \overset{*}{\Rightarrow} V \cdots, V \in V_T \cup V_N \right\} \cup \left(\alpha \overset{*}{\Rightarrow} \varepsilon \,?\{\varepsilon\} : \phi \right).$$

***Follow* set:**

Given any grammar symbol $X \in V_T \cup V_N$, $Follow(X)$ is defined as

$$Follow(X) = \left\{ V \middle| S \overset{*}{\Rightarrow} \cdots XV \cdots, V \in V_T \cup V_N \right\} \cup \left(S \overset{*}{\Rightarrow} \cdots X \,?\{\#\} : \phi \right).$$

Thus, in this case both *First*() and *Follow*() may include nonterminals. The algorithms to compute these two sets can be easily constructed according to the above definitions.

As an example, we briefly discuss the construction of SBLR (Simple BLR) parsing tables. Similarly to SLR parsing, for a given grammar *G*, by introducing an augmented grammar *G*', effective item sets, *Closure*( ), *go*() function and regular group of item sets $C = \{I_0, I_1, I_2, \ldots, I_n\}$, and using the *First*() and *Follow*() sets defined above, we have the following algorithm of constructing an SBLR parsing table.

Algorithm of SBLR(0) parsing table construction

Input: an augmented grammar *G*';

***Output*:** a parsing table of *G*' *Action*[];

***Steps*:**

for each effective item set I_i of C **do begin**
  if $A \to \alpha \cdot X \beta \in I_i$, $X \in V_T \cup V_T$ **and** $go(I_i, X) = I_j$ **then begin**
    $Action[i, X]$:= shift j ;
  end
  else if $A \to \alpha \cdot \in I_i$, $A <> S'$ **then begin**
    for all $X \in Follow(A)$ **do begin**
      $Action[i, X]$:= reduce $A \to \alpha$;
    end
  end
  else if $S' \to S \cdot \in I_i$ **then begin**
    $Action[i, \#]$:= *ACCEPT* ;
  end
  else $Action[i, X]$:= *ERROR* ;
end

If we redefine the *Closure*() set to include nonterminals as lookahead symbols and extend the *go*() function accordingly, we can further improve the above parsing tables with lookahead parsing, which is similar to the improvement from SLR to LR(1) parsing. Furthermore, by introducing the closure kernels of those items sets, an LALR like parsing table may also be constructed for the same grammar. (See [8] for more details.)

5. Use of the parsing technique

The parsing method proposed here can be used to handle many grammatical ambiguities that may introduce LR conflicts. As a typical example case, if a grammar G has the segments of productions: $S \to AC \mid BD$, $A \to xy$, $B \to xy \mid \ldots$, where *first*(C) and *first*(D) have some common terminals, then the usual shift-reduce parsing would encounter conflicts, since the parser has to make decision to either reduce to A or B, or to shift the input. In BSR parsing, the parser can have the third choice to postpone the decision of xy reduction, and go on to handle the sequential tokens, and then to make decision of reduction to A or B based on whether the latest result is C or D.

To achieve this controllability, the parsing table needs to be properly constructed. Currently only construction of simplified parsing tables is used, as discussed above. Nevertheless, with appropriate use of the intermediate buffer, many nontrivial grammatical ambiguities can be handled in a systematical way, providing efficient parsing compared to corresponding LR parsers.

6. Summary and future work

Buffered shift-reduce parsing, by postponing the decisions of shift/reduce actions using a symbol buffer, provides more control and flexibility of ambiguity handling, and is applicable to parsing a class of ambiguous grammars that may introduce typical conflicts to LR parsing. This paper presents the basic ideas of the technique and the construction of parsing tables in some simplified cases. There remains some work that is worth further considerations, including parsing tables of a few more general cases (not necessarily all BSR grammars in practical use), and the extension of a Tomita algorithm like BSR parsing version, where shared parsing stack and buffer can be employed for grammars with multiple parsing table entries in a breadth-precedence way.

References

[1] Dick Grune and Ceriel Jacobs, Parsing Techniques – A Practical Guide, Vrije Universiteit, Amsterdam, 1998. (Available online at ftp://ftp.cs.vu.nl/pub/dick/PTAPG/BookBody.ps.gz)

[2] Bison, the GNU Compiler Compiler (Version 1.28), 1999. Free Software Foundation, Inc. Available at www.gnu.org.

[3] Masaru Tomita, Efficient Parsing for Natural Language – A Fast Algorithm for Practical Systems, Kluwer Academic Publishers, 1986.

[4] Tao Xiaopeng, Machine translation oriented Chinese syntactical rules and parsing. MA thesis, CS Dept., Peking University, 1995. (陶晓鹏，面向机器翻译的汉语句法规则和自动分析，硕士论文，北京大学计算机系, 1995。)

[5] Bing Swen, BYACC: Design of a compiler generator. Internal documentation, ICL, Peking Univsersity, 1998. (孙宾，BYACC：一种编译程序生成程序的设计，内部资料，北京大学计算语言学研究所，1998。)

[6] M. P. Marcus, A Theory of Syntactic Recognition for Natural Language, P.H. Winston and R.H. Brown (eds), Artificial Intelligence: An MIT Perspective, Vol. 1, MIT Press, 1979.

[7] Qian Feng, Introduction to computational linguistics. Xuelin Press, 1990. (钱锋，计算语言学引论，学林出版社，1990。)

[8] Bing Swen, Shift-Reduce/Putback Parsing (in Chinese). Technical Report, ICL Jan. 1999, CS Dept., Peking Univsersity. URL http://icl.pku.edu.cn/bswen/nlp/bsPhdMidExam.html.