Pergamon

PII: S0096-0551(96)00015-X

Comput. Lang. Vol. 22, No. 4, pp. 225–235, 1996
© 1997 Elsevier Science Ltd. All rights reserved
Printed in Great Britain
0096-0551/97 $17.00 + 0.00

BUILDING EFFICIENT INCREMENTAL LL PARSERS BY AUGMENTING LL TABLES AND THREADING PARSE TREES

WARREN X. LI

Software Verification Research Center, Department of Computer Science, The University of Queensland, Queensland 4072, Australia

(Received 3 July 1995; revision received 7 August 1996)

Abstract—Incremental parsing is widely used in language-based editors and incremental compiling and interpreting environments. Re-parsing of modified input strings is the most frequently performed operation in these environments. Its efficiency can greatly affect the success of these environments. This paper describes the introduction of a threaded link in parse trees and an additional *distance* entry in LL parse tables to support minimal LL(1) re-parsing. These enhancements are then used to produce an efficient incremental LL parser. © 1997 Elsevier Science Ltd

1. INTRODUCTION

The development of language-based editors has drawn increased attention to the problem of incremental parsing. Incremental parsers attempt to minimize the re-parsing process when a change is made. When incremental parsing is employed in language-based editors, the rapid feedback of the change is crucial to the success of such editors. A number of algorithms have been proposed for LR incremental parsers [1–4], but few have been proposed for LL incremental parsers [5, 6]. However, both of the LL-based algorithms in these papers attempt to minimize the re-parsing based on the original parse tree and the parse table. Both algorithms have a limited success on their attempt to maximise the re-use of the existing parse tree.

In this paper, we introduce a threaded link in parse trees (termed *threaded parse trees*) and an additional *distance* entry in LL parse tables (termed *augmented LL parse tables*). A more efficient incremental LL parser based on the threaded parse tree and the augmented LL parse table has then been developed. This parser has achieved the maximal re-use of the previously established parse tree and the earliest possible detection of unnecessary re-parsing. In particular, The algorithm presented works optimally over block and right-recursive list structured languages since the subtrees representing these constructs are considered directly for re-use.

Let $G = (N, T, P, S)$ be an unambiguous context-free grammar with N representing the set of nonterminals, T the set of terminals, P the set of productions, and S the start symbol (we adopt the notation used in [7, 8]). We assume that G is a $-augmented grammar. If not, we can build the $-augmented grammar $G' = (N\cup\{S'\}, T\cup\{\$\}, P\cup\{S' \rightarrow S\$\}, S')$ where $ is the symbol representing end of input.

For an input string xyz generated by G, we have $S \Rightarrow^* xyz$, $x,y,z \in T^*$. Incremental parsers attempt to perform a minimal number of parsing steps when the input string xyz is changed to $xy'z$. In this paper we discuss the re-parsing process in the presence of a parse tree and an LL parse table. Although our motivation in developing an efficient LL parser is for its use within language-based editors, our approach is not limited to such editors; the incremental parser can be extended to incremental compiling and interpreting environments.

We use the following LL(1) grammar, $G1$, in examples throughout the paper.

(1) $S \rightarrow E\ \$$
(2) $E \rightarrow T\ R$
(3) $R \rightarrow +\ T\ R$

(4) $R \rightarrow \lambda$
(5) $T \rightarrow P\ V$
(6) $V \rightarrow *\ P\ V$
(7) $V \rightarrow \lambda$
(8) $P \rightarrow a$
(9) $P \rightarrow b$
(10) $P \rightarrow c$
(11) $P \rightarrow (E)$

2. AUGMENTING LL TABLES AND THREADING PARSE TREES

To obtain an efficient LL incremental parser, we augment LL parse tables and parse trees. A metric, termed *distance*, is added to each valid entry of an LL parse table and a link, termed *threaded link*, is annexed to each node of a parse tree.

The LL parse table LLT is of the form

$$LLT{:}\{N \cup T\} \times T \rightarrow E \cup \{\ error,\ success,\ match\}$$

where E is the set of all production numbers. If A is a nonterminal and t is a lookahead token, $LLT(A,t)$ tells which production to predict. If A is a terminal, $LLT(A,t)$ tells if A matches t or not. The *errors* and *success* in $LLT(A,t)$ indicate syntax errors and terminations respectively.

Definition 1. *The augmented LL parse table ALLT is defined as the form*

$$ALLT{:}\{N \cup T\} \times T \rightarrow E \times D \cup \{(match,0),(success,0),(error,0)\}$$

where E is the set of all production numbers and D is a set of integers, termed distances. For each entry $(p,d) = ALLT(A,t)$, p is a production number same as in a traditional LL(1) parsing table, and the distance d is calculated by:

$$d(A,t) = \begin{cases} 0; & \textit{if p is match, success or error} \\ -1; & \textit{if } t \in First(A) \textit{ and } p \textit{ is } A \rightarrow \lambda \\ d(B,t)+1; & \textit{if } t \in First(A),\ t \in First(b) \textit{ and } A \rightarrow B\ldots \in P \end{cases}$$

That is, the distance d is the number of minimum derivations applied to extend the A to a string that contains the terminal t as its leftmost symbol:

$$A \Rightarrow X_1X_2\ldots X_n \Rightarrow B_1B_2\ldots B_mX_2\ldots X_n \Rightarrow \ldots \Rightarrow t\ldots B_2\ldots B_mX_2\ldots X_n.$$

Note that for a given LL(1) grammar, the B in the distance definition is unique. This is because if we have another symbol, say C ($C \neq B$), that has $t \in First(C)$ and $A \rightarrow C\ldots \in P$, then we have ambiguous on $LLT(A,t)$ (that can predict both $A \rightarrow B\ldots$ and $A \rightarrow C\ldots$). It contradicts the fact that the grammar is an LL(1) grammar. For the same reason, we can conclude that for a given LL(1) grammar, the distance $d(A,t)$ is unique.

The augmented LL table for the grammar $G1$ is presented in Table 1.

The generation of the augmented LL table can be either made from an existing LL table or produced together with the LL table by an LL table generator such as LLGen [9] and LLama [10].

Table 1.

	+	*	a	b	c	(	)	$
S			(1,4)	(1,4)	(1,4)	(1,4)		
E			(2,3)	(2,3)	(2,3)	(2,3)		
R	(3,1)						(4, − 1)	(4, − 1)
T			(5,2)	(5,2)	(5,2)	(5,2)		
V	(7, − 1)	(6,1)					(7, − 1)	(7, − 1)
P			(8,1)	(9,1)	(10,1)	(11,1)		
+	(M,0)							
*		(M,0)						
a			(M,0)					
b				(M,0)				
c					(M,0)			
(						(M,0)		
)							(M,0)	
$								(S,0)

The M, S denote match and success in short, and nothing for error.

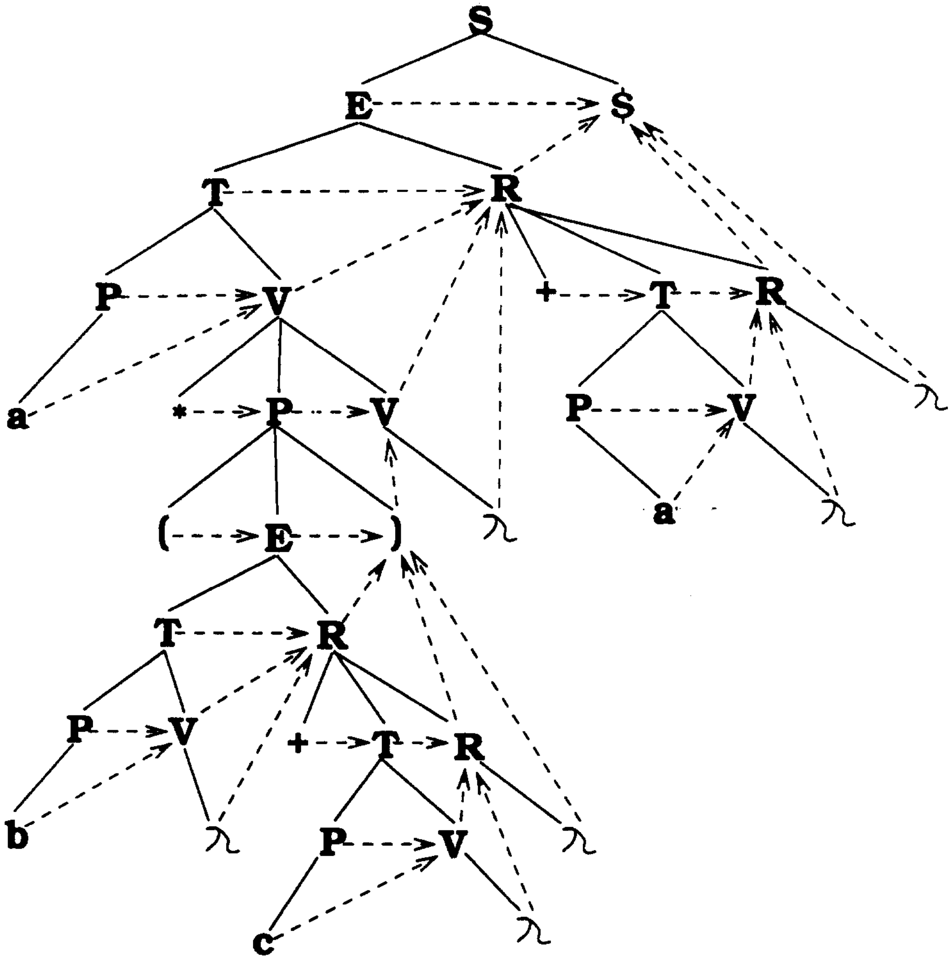

Fig. 1. The threaded LL parse tree of $a*(b+c)+a$.

The generation of the ALLT from an existing LL table simply requires the consecutive applications on the first nonterminal in the derivations. The second can be implemented by simply modifying the LL table generator.

Definition 2. *A threaded LL parse tree is a parse tree in which there is a pointer on each node linked to either*

(i) its right sibling (if one exists), or

(ii) the right sibling of its closest ancestor that has a right sibling (if such an ancestor exists), or

(iii) NIL.

Figure 1 is an example of the threaded LL parse tree which represents the sentence $a*(b+c)+a$ of grammar $G1$.

We use the notation $\Gamma(N)$, the node pointed by the threaded link of N, to denote the *threaded node of* N. From Fig. 1 we observe that $\Gamma(b) = V$ and $\Gamma(V) = R$. And the notation $\Gamma^i(N)$ is defined as follows:

$$\Gamma^0(N) = N$$

$$\Gamma^i(N) = \Gamma(\Gamma^{i-1}(N)), \quad i \geq 1$$

At the node b, we have $\Gamma^0(b) = b$, $\Gamma^1(b) = \Gamma(b) = V$ and $\Gamma^2(b) = R$). We also denote $\Gamma\dagger(N)$ (the *threaded links of* N) as the concantenation:

$$\Gamma^\dagger(N) = \Gamma^1(N)\cdot\Gamma^2(N)\ldots\Gamma^k(N);\ \Gamma^j(N) \neq NIL,\ j \leq k$$

$$\Gamma^{k+1}(N) = NIL.$$

At the node b, we have $\Gamma\dagger(b) = VR)VR\$$. Traditional LL parsers which employ a parsing stack, push the right hand side of each predicted production onto their stack. The stack always retains 'future' parsing symbols. Observing the threaded links described here, it can be found that the $\Gamma\dagger(N)$ represents the content of the traditional parse stack. In stack-based parsers, the content of the parse stack after parsing "a *(b" is "VR)VR\$" which are also represented by the threaded links $\Gamma\dagger(b)$. $\Gamma^1(N)$ simply represents the top of the parse stack.

If a production $P \rightarrow P_1P_2 \ldots P_n$ is applied at a node P, the threaded link $\Gamma(P_i)$ of each child P_i is simply obtained from

$$\Gamma(P_i) = P_{i+1}, 1 \leq i \leq n-1;$$

$$\Gamma(P_n) = \Gamma(P).$$

3. INCREMENTAL LL PARSING

When considering a change from a syntactically correct string xyz (with a parse tree T) to $xy'z$, incremental parsers need to construct a new parse tree T' corresponding to the string $xy'z$. Either y or y' can be empty. An empty y stands for an insertion while an empty y' stands for a deletion.

The simplest solution to this problem is to parse the whole string $xy'z$ from scratch. Obviously this is a time-consuming process and is not what incremental parsers intend to do. Since x is the shared prefix of xyz and $xy'z$, there is no need to re-parse the string x, and significant time is reduced by skipping it and starting with string $y'z$. Since z is also the shared suffix of both strings, the maximum re-use of z is what the incremental parsers are concerned with.

3.1. *The re-use of parse trees*

Since x is the common prefix and z is the common suffix for xyz and $xy'z$, the subtrees related to x and z need to be considered for re-use. To do this, we need to split the tree T into the subtrees related to x and z.

Let us consider the parsing of x. Since x is a valid prefix for xyz, we have $S \Rightarrow^* xA_1 \ldots A_n$. $S \Rightarrow^* xA_1 \ldots A_n$ represents the derivation from the start symbol S at the moment the string x has just been parsed, and $A_1 \ldots A_n$ $(A_j \in T \cup N)$ is the content of parsing stack at that moment. The incomplete parse tree built during the parsing of x is called *X-partial-tree*. Since the *X-partial-tree* is shared by both xyz and $xy'z$, we can re-use it without re-parsing.

Let t be the last terminal of x. The threaded links of t, $\Gamma\dagger(t)$, represent the $A_1 \ldots A_n$, i.e., $\Gamma\dagger(t) = A_1 \ldots A_n$. Thus we can obtain the *X-partial-tree* from the original parse tree T by splitting T on the thread links $\Gamma\dagger(t)$.

Let t be "(" in the example of the parse tree in Fig. 1, we have the *X-partial-tree* in Fig. 2, in which $A_1 \ldots A_n = E)VR\$$.

We can get the subtrees related to z with threaded links. Let l (possibly epsilon) be the last terminal of the string y (l becomes t if y is empty). The threaded links $\Gamma\dagger(l)$ represent subtrees which produce the string z. Let

$$\Gamma^{\dagger}(l) = B_1B_2 \ldots B_m$$

we have $B_1B_2 \ldots B_m \Rightarrow^* z$.

The subtrees $B_1B_2 \ldots B_m$ are called the *Z-subtrees*. Let l be "(" in the example of the parse tree in Fig. 1, we have the *Z-subtrees* in Fig. 3, in which $B_1B_2 \ldots B_m = E)VR\$$. Note that the example given here is the case $l = t$ (i.e. y is empty), therefore $B_1B_2 \ldots B_m = A_1 \ldots A_n$. For a general modification (y is not empty), $B_1B_2 \ldots B_m$ is different from $A_1 \ldots A_n$.

3.2. *Outline of incremental parsing*

Since x is the common prefix for both xyz and $xy'z$, there is no need to re-parse the *X-partial-tree*. We start to parse $y'z$ with the *X-partial-tree*. Obviously, the string y' has to be fully parsed. When parsers begin to parse string z, they try to re-use as many *Z-subtrees* as possible.

Let *new* be a pointer to the node that is being built, and *old* be a pointer to the node that is considered for re-use in the *Z-subtrees*.

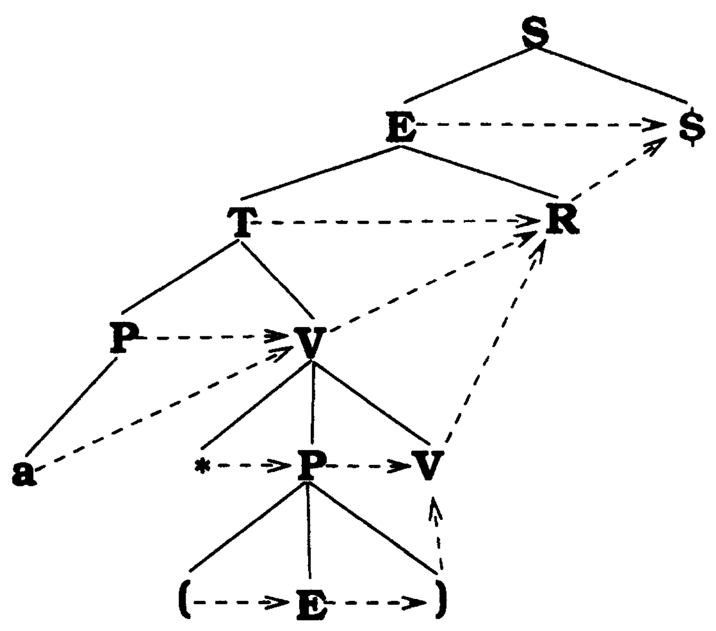

Fig. 2. The *X-partial-tree*.

Initially *new* is set to A_1. The parser processes y' as a traditional non-incremental parser. But threaded links of newly created nodes are built as the parsing of y' proceeds.

After y' is parsed, the parser tries to maximize the re-use of the *Z-subtrees*. We set *old* to be B_1 initially. The *token* is also defined as the current lookahead token (or terminal), setting to be leftmost terminal of B_1 initially.

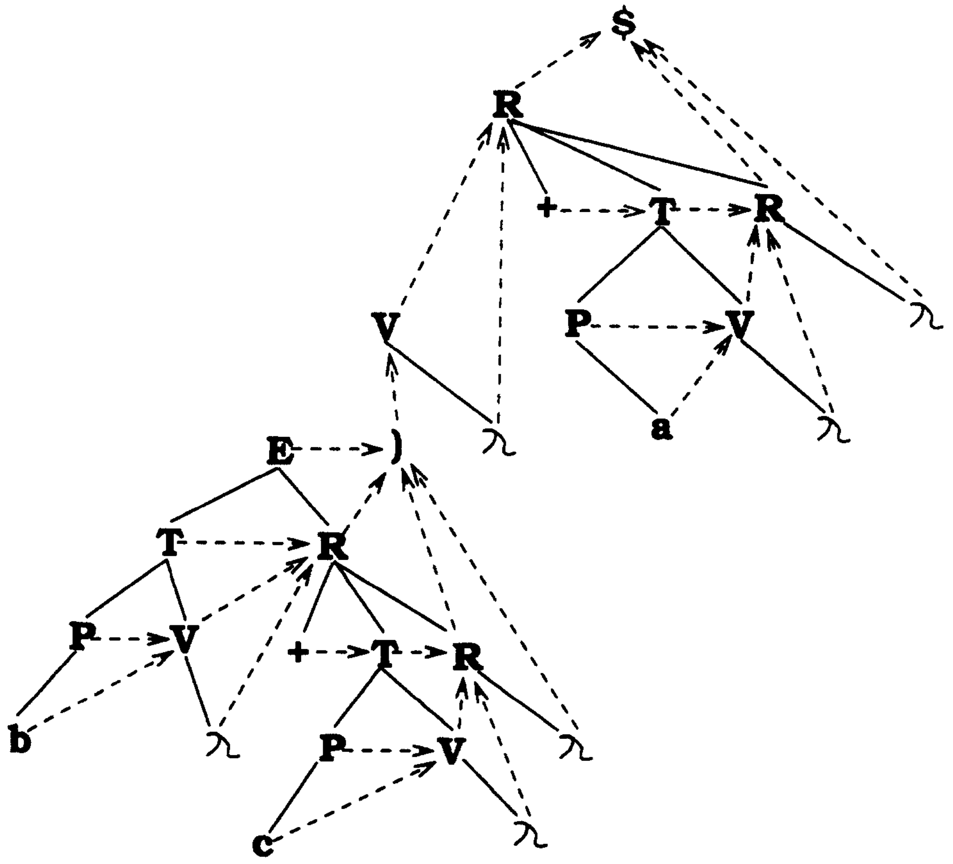

Fig. 3. The *Z-subtrees*.

The parser first checks if the *token* is a valid lookahead for *new*. If not, a syntactic error has occured and reported. If the *token* is a valid lookahead terminal, the re-use of a subtree (at least the *token* leaf node) is guaranteed. If the node pointed by *new* (we simply call the node *new*) has the same syntax symbol (terminal or nonterminal) as the node *old* (we say they match), the re-parsing of the subtrees of the node *old* is avoided, and the subtrees of the node *old* is copied to the node *new*.

If the node *new* does not match the node *old*, there is still a chance that partial matching may occur. The partial matching may occur in the following cases:

- the node *new* may match the descendant of the node *old*;
- the descendant of the node *new* may match the node *old*;
- the descendant of the node *new* may match the descendant of the node *old*.

The *distance* in the augmented LL table can be used in determining and maximizing the matching. Let $ALLT(new, token) = (P_1, d_1)$ and $ALLT(old, token) = (P_2, d_2)$. Note that *new* matching *old* implies $d_1 = d_2$. That is, the matching can only occur when $d_1 = d_2$. If the distances d_1 and d_2 are different, the three partial matching mentioned above should be tried, that is, by reducing the larger distance d_i $(i = 1,2)$ to be the same as the other.

This is another case we have not mentioned above, that is, $d_1 = -1$. The case indicates that an epsilon production is applied, the parse tree is built as in a normal non-incremental parser.

We thus outline, in brief, the steps of the incremental parsing process as follows.

```
(i) Split the parse tree $T$ (representing $xyz$) into an X-partial-tree (representing $xA_1 \ldots A_n >$ ),
and Z-subtrees ($B_1 \ldots B_m$).

    Let new point to $A_1$; old to $B_1$.

    (ii) Parse the string $y'$.
    (iii) {Analysis of the string $z$}
        Get a token (the leftmost terminal of $B_1$ initially);
        let $(p_1, d_1) = ALLT(new, token)$; $(p_2, d_2) = ALLT(old, token)$.
        (a) If new-NIL then success and stop;
            If $p_1 = error$ (token is not a valid lookahead), then report a error;
            if $d_1 = -1$ (an epsilon production is applied) then goto step (d);
            if $d_1 < d_2$ then goto step (b);
            if $d_1 > d_2$ then goto step (c);
            if $d_1 = d_2$ then goto step (d);
        (b) Move old to its first child; $(p_2, d_2) = ALLT(old, token)$;
            repeat this step until $d_2 = d_1$;
            goto step (a).
        (c) Build the tree structure of the node new with the production $p_1$;
            move new to its first child; $(p_1, d_1) = ALLT(new, token)$
            repeat this step until $d_1 = d_2$
            goto step (a).
        (d) If they match (new and old point to a node that has same syntax symbol);
            copy all the subtree of old to new;
            move new to $\Gamma(new)$; old to $\Gamma(old)$;
            get a new token from the leftmost terminal of old;
        otherwise (not matching)
        build the tree structure of the node new with the production $p_1$;
        move new to its first child; old to its first child;
      let $(p_1, d_1) = ALLT(new, token)$; $(p_2, d_2) = ALLT(old, token)$;
      goto step (a).
```

Splitting a parse tree to an *X-partial-tree* and *Z-subtrees* greatly simplifies the algorithm in concept. On the other hand, it leads to some redundant operations, that is, unnecessary splitting on rearmost nodes of $A_1 \ldots A_n$ and consequent redundant copies on these nodes. This is because a modification from xyz to $xy'z$ often affects only areas local to certain subtrees. The algorithm

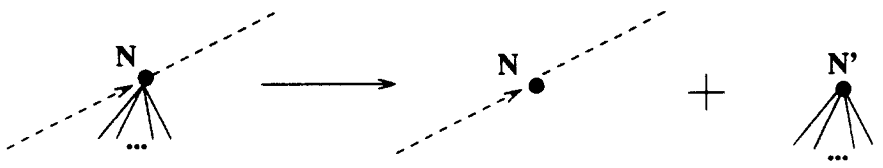

Fig. 4. Breaking the node N.

in the following section will not split the parse tree at the beginning of the algorithm. Instead, the splitting of A_j only occurs if necessary at the time that the parser processes the node. In this way, the rearmost nodes of $A_1 \ldots A_n$ will not be split, and there will be no redundant copies of these nodes, and consequently the algorithm can stop re-parsing earlier. The efficiency of the algorithm is then improved.

3.3. *The incremental LL parsing algorithm*

In the following, we first define attributes for nodes in a threaded LL parse tree, variables and functions which will be used in the algorithm. Then we give the algorithm in detail and an example.

Each node N in the threaded LL parse tree has following attributes.

$\upsilon(N)$: grammar symbol of N;
father(N): a pointer to its father;
firstchild(N): a pointer to its first child;
$\Gamma(N)$: the threaded link of N.

Note that it does not require more attributes than a normal parse tree structure. In this structure, the Γ is used instead of a pointer to its sibling.

We define the following variables:

new: a pointer to the node currently being parsed;
old: a pointer to the node in the *Z-subtrees* which is considered for re-use;
token: the lookahead token from $y'z$;

We use the following functions for the algorithm:

ALLT(A, t): the augmented LL table; returning a production number and a distance, (p, d), or error for a given grammar symbol A (row) and a terminal t (column).

empty_node(): making an empty node; returning a pointer to the node.

next_input_token(): getting next token from y'.

leftmost_leaf(N): getting the leftmost non-epsilon token of the tree N.

breakup(N): breaking the node N from the original tree T while the N is one of the threaded nodes $A_1 \ldots A_n$ (depicted in Fig. 4). The splitted node will be considered for re-use. Let the production at N be $N \rightarrow N_1 \ldots N_m$.

N'	= *empty_node*();
$\upsilon(N')$	$= \upsilon(N)$;
$\Gamma(N')$	$= \Gamma(N)$;
firstchild(N')	= *firstchild*(N);
firstchild(N)	= *NIL*;
father(N_i)	$= N', 1 \le i \le m$;

build(*new*, p): build the immediate children of the node *new* from production $p(N \rightarrow N_1 \ldots N_n)$.

$N_{,i}$	= *empty_node*(), $1 \le n$;
father(N_i)	$= \mathrm{N}, 1 \le i \le n$;
firstchild(*new*)	$= \mathrm{N}_1$;
$\Gamma(N_i)$	$= \mathrm{N}_{i+1}, 1 \le i \le n-1$;
	$= \Gamma(new), i = n$;
$\upsilon(N_i)$	$= \|N_i\|, 1 \le i \le n$;

copy(*old*, *new*): copying all the subtrees of *old* to *new*. After copying, the threaded links on the rightmost descendants have to be changed to be the same as the threaded link of *new*, i.e. Γ(*new*). The complexity depends on the length of the route from *new* to the rightmost leaf of the subtree.

The algorithm initializes *new* to A_1, the first on the threaded links $A_1 \ldots A_n$, and starts to parse the y'. The traditional non-incremental parsing applies on y', but threaded links are built during the parsing. Also, while the current parsing node *new* is among the threaded links $A_1 \ldots A_n$ (in the

```
{parsing $y'$}
$new = \Gamma(t)$; {t is last non epsilon terminal of $x$}
$token = next\_input\_token()$;
while $token \neq nil$ do
   if $firstchild(new) \neq nil$ then $breakup(new)$ fi;
   $(p, d) = ALLT(new, token)$;
   if $p = error$ then report error and stop fi;
   if $p = match$ then
      $new = \Gamma(new)$;
      $token = next\_input\_token()$;
   else
      $build(new, p)$;
      $new = firstchild(new)$;
   fi
od

{parsing $z$}
$old = B_1$; {$\Gamma(l)$ where $l$ is the last terminal of $y$}
$token = leftmost\_leaf(old)$;
$(p_1, d_1) = ALLT(new, token)$;  $(p_2, d_2) = ALLT(old, token)$;
while $new \neq old$ do {loop until matching}
   if $p_1 = error$ then report error and stop fi;
   if $firstchild(new) \neq nil$ then $breakup(new)$ fi;
   if $d_1 = -1$ then
      $build(new, p_1)$;
      $new = \Gamma(new)$;
      $(p_1, d_1) = ALLT(new, token)$;
   else if $d_1 < d_2$ then
      $old = firstchild(old)$;
      $(p_2, d_2) = ALLT(old, token)$;
   else if $d_1 > d_2$ then
      $build(new, p_1)$;
      $new = firstchild(new)$;
      $(p_1, d_1) = ALLT(new, token)$;
   else if $d_1 = d_2$ and $v(new) = v(old)$ then
      $copy(old, new)$;
      $old = \Gamma(old)$;  $new = \Gamma(new)$;
      $(p_1, d_1) = ALLT(new, token)$;  $(p_2, d_2) = ALLT(old, token)$;
      $token = leftmost\_leaf(old)$;
   else {$d_1 = d_2$ and $v(new) \neq v(old)$ }
      $build(new, p_1)$;
      $old = firstchild(old)$;  $new = firstchild(new)$;
      $(p_1, d_1) = ALLT(new, token)$;  $(p_2, d_2) = ALLT(old, token)$;
   fi
od
```

Fig. 5. Algorithm.

Table 2. The re-parsing steps

E)VR\$	a + E)VR\$	predict 2
TR)VR\$	a + E)VR\$	predict 5
PVR)VR\$	a + E)VR\$	predict 8
aVR)VR\$	a + E)VR\$	match
VR)VR\$	+ E)VR\$	predict 7
R)VR\$	+ E)VR\$	predict 3
+ TR)VR\$	+ E)VR\$	match
TR)VR\$	E)VR\$	$d_1(2) < d_2(3)$ split
TR)VR\$	TR)VR\$	match (re-use)
R)VR\$	R)VR\$	match (re-use)
)VR\$	)VR\$	terminate

case that *firstchild(new)* = *nil*, the *breakup* function is invoked to split a subtree from the previously established parse tree T.

Before parsing z, *old* is initially set to B_1, the first node of threaded links $B_1 \ldots B_m$. The parser starts to parse z with the hope of re-using the *Z-subtrees* until no more re-parsing is needed. The condition *new* = *old* indicates the rest of the tree is shared by xyz and $xy'z$, therefore there is no more parsing needed.

If there is no syntactic error, the difference between the distance d_1 for *new* and the distance d_2 for *old* for a given lookahead *token* is then tested to determine next step of the algorithm. There are cases, i.e., $d_1 = d_2$, $d_1 > d_2$, $d_1 < d_2$ and $d_1 = -1$.

(i) If $d_1 < d_2$, *old* moves down to its first child (d_2 decreased by one) in order to make $d_1 = d_2$.
(ii) If $d_1 > d_2$, the immediate children of *new* are built according to the production p in *ALLT* (*new*, *token*). Then *new* moves down to its first child (d_1 decreased by one) and one step closer to have $d_1 = d_2$.
(iii) If $d_1 = d_2$, in this case, the matching will be successful if $v(new) = v(old)$, and the copying from the subtrees of *old* to *new* is then done. If $v(new) \neq v(old)$, the immediate children of *new* are built according to the production p_1 in $(p_1, d_1) = ALLT(new, token)$. And then both *new* and *old* move to their first children, which $d_1 = d_2$.
(iv) If $d_1 = -1$, an epsilon production is applied and the subtree with the production $A \rightarrow \lambda$ is built as in a traditional non-incremental parser.

The algorithm is shown in Fig. 5.

3.4. An example

We take the previous grammar g1 and the string $a*(b + c) + a$ (Fig. 1) as an example. If the string is modified to $a*(a + b + c) + a$, i.e. inserting y' = "$a +$ " after the open parenthesis, we have

$$x = \text{“}a*(\text{”}$$
$$y = NIL$$
$$y' = \text{“}a + \text{”}$$
$$z = \text{“}b + c) + a\text{”}.$$

The *X-partial-tree* and the *Z-subtrees* are depicted in Figs 2 and 3, respectively. The re-parsing steps are in Table 2.

After incrementally parsing, the parse tree of the new string $a*(a + b + c) + a$ is depicted as Fig. 6. The parts under the curve L are only parsed and the two shadow areas C_1 and C_2 are re-used from the previous parse tree.

3.5. Correctness and efficiency

Firstly, we look at the *X-partial-tree*. Since the *X-partial-tree* is the parse tree when x has just been parsed, the *X-partial-tree* is a valid partial tree for both xyz and $xy'z$. The algorithm starts with the *X-partial-tree*, and adopts the traditional table-driven LL(1) parsing to parse the string y'. So we only need to prove the parsing process on *Z-subtrees* (z).

In the algorithm, the variable *old* always points the frontier subtree of the remaining *Z-subtrees* for re-use, and *token* points the leftmost leaf terminal of the subtree pointed by *old*. The algorithm first checks if the *token* is a valid lookahead terminal for *new*. The algorithm then uses the principle

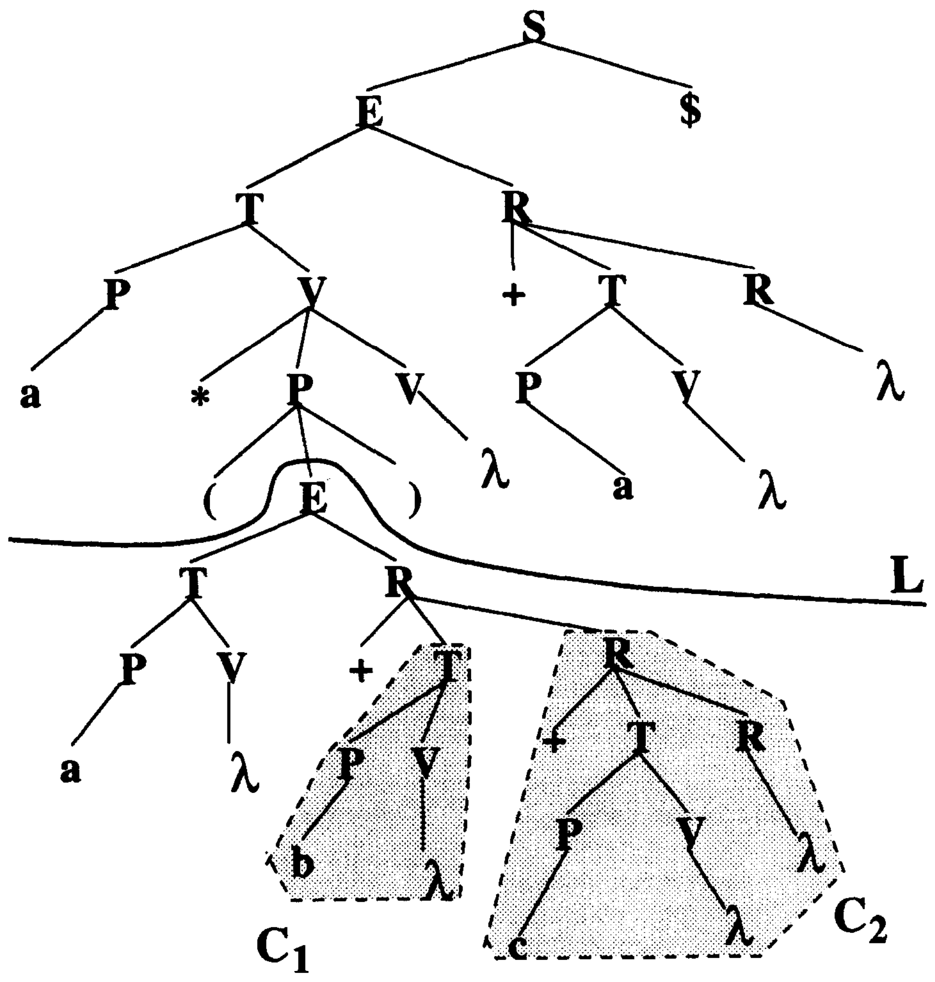

Fig. 6. Re-parsing after inserting $a+$ in $a^*(b+c)+a$.

that, for a given nonterminal A and a given terminal t, the distance d in $ALLT(A,t)$ is unique. That is, the re-use of a subtree can only happen when the subtree *old* has the same distance as the node *new*. The algorithm follows this principle to reduce one of the distances. Apart from the distances, the algorithm is based on the traditional LL(1) parsing. The distance $d_1 = -1$ indicates that an epsilon production is applied. For this case the algorithm uses the traditional LL parsing to build up the parse tree. So we have proven the algorithm is correct.

As you can see from the implementation, a threaded link Γ is added to substitute the right sibling link which a normal parse tree must have. Thus there are not extra spaces required in the threaded LL parse tree. So it is space efficient.

The variable *old* always points to the largest subtree split from the original parse tree T. In other words, the largest possible subtrees are always tried for re-use. Therefore the algorithm maximizes

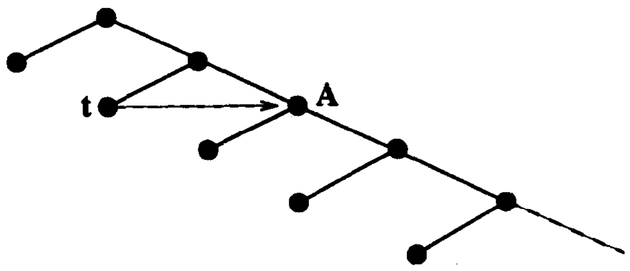

Fig. 7. The algorithm handles the right-recursive list.

the re-use of the previous parse tree. The algorithm also detects the earliest possible termination condition so that unnecessary re-parsing can be avoided. So it is also time efficient.

4. CONCLUSIONS

In this paper we have introduced an augmented LL table and a threaded LL parse tree to support efficient incremental LL parsing. Finally a space and time efficient algorithm has been presented. It can be widely used in language-based editors and incremental compiling and interpreting environments.

The incremental parser is built based on the presence of the threaded parse tree and the augmented LL parse table. The two augmentations have improved the efficiency of incremental parsing. In fact, the two augmentations act individually in the algorithm, so an incremental parser with either augmentation can be built with a minor modification to the above algorithm though the efficiency will not be as good as the one with the two augmentations.

It should be noted that this algorithm can easily extend to multiple modifications. If we let input string be $xy_1z_1...y_nz_n$, the multiple modifications mean the changes on every y_k, $1 \leq k \leq n$, i.e., $xy_1'z_1y_2'z_2...y_n'z_n$. If we consider the further splitting on $A_1...A_n$, we can have multiple *Z-subtrees* for $z_1z_2...z_n$ to be considered for re-use. A minor modification of the algorithm can apply to the multiple modifications.

The algorithm particularly works very well on languages with a large number of block structures and lists (note that LL's lists are right recursive). This is because the algorithm takes blocks and lists as units to be considered for re-use. Looking at the tree structure of right-recursive list in Fig. 7, we can get the part of the list, the subtree rooted at A, as a unit when the t is modified.

The algorithm has been used as a key element in our implementation of *TextSyn* [11, 12] language-based editing system. *TextSyn* performs syntactic and semantic analysis while editing a program. The algorithm is employed incrementally to analyse the program, and to report any syntactic inconsistencies. The success of using this algorithm in this system has been proven its correctiness and efficiency. The response time of performing an incremental parsing is very fast and hardly noticed by users during editing in this environment.

REFERENCES

1. A. Celentano. Incremental LR parsers. *Acta Informatica*, 1978, **10**, 307–321.
2. P. Degano, S. Mannucci and B. Mojana. Efficient incremental LR parsing for syntax-directed editors. *ACM Transactions on Programming Languages and Systems*, 1988, **10**, 345–373.
3. C. Ghezzi and D. Mandrioli. Augmenting parsers to support incrementality. *Journal of the Association for Computing Machinery*, 1980, **27**, 564–579.
4. Jalili, F. and Gallier, J., Building friendly parsers. In *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages*, 1982.
5. N. Delisle, M. Schwartz and V. Begwani. Incremental compilation in magpie. *SIGPLAN Notices*, 1982, **19**, 122–130.
6. A. Murching, Y. Parsad and Y. Srikant. Incremental recursive descent parsing. *Computer Languages*, 1990, **15**, 193–204.
7. Aho, A. and Ullman, J., *The Theory of Parsing, Translation and Compiling*, Vol. 1 and 2. Prentice-Hall, Englewood Cliffs, NJ, 1972 and 1973.
8. Aho, A., Sethi, R. and Ullman, J., *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.
9. D. Grune and C. Jacobs. A programmer-friendly LL(1) parser generator. *Software-Practice and Experience*, 1988, **18**, 29–38.
10. Holub, A., *Compiler Design in C*. Prentice-Hall International, 1990.
11. Li, W. and McDonald, C., A specification-driven language editor. In *Proceedings of Fifteenth Australian Computer Science Conference*, 1992, pp. 561–572.
12. Li, W., Towards generating practical language-based editing systems. Ph.D. Thesis. Department of Computer Science, The University of Western Australia, 1995.

About the Author—Warren Li received his M.Sc. from Chongqing University, China and Ph.D. from The University of Western Australia, Australia in 1988 and 1995, respectively. He also worked in the computer industry for a number of years. He is currently a Research Fellow at the Software Verification Research Center, The University of Queensland, Australia. His research interests include: Programming Languages and Implementation, Programming Environment and Software Engineering.