Incremental Attribute Evaluation of LR-attributed grammars Using Space-Efficient Data Structure

Hisashi Nakai[1], Masataka Sassa[2], Hiroaki Kameyama[3] & Ikuo Nakata[4]

1: University of Library and Information Science,
1-2 Kasuga Tsukuba, Japan
nakai@ulis.ac.jp
2: Tokyo Institute of Technology,
sassa@is.titech.ac.jp
3: University of Tsukuba,
kame@lang.esys.tsukuba.ac.jp
4: Faculty of Computer and Information Sciences, Hosei University
nakata@ulis.ac.jp

Abstract

Incremental attribute evaluation of one-pass attribute grammars (AGs) has not yet been fully investigated. However, considering the lightness of one-pass AGs, combining incremental evaluation and parsing in one-pass AGs may bring about a time- and space-efficient language processor.

In this paper, an incremental attribute evaluation method based on LR-attributed grammar, a class of one-pass attribute grammar, is described. First we propose a method of incremental attribute evaluation using a parse tree. Then we present a space-optimized incremental attribute evaluation method using a more compact data structure whose number of nodes is the same as that of tokens. We also describe an optimization to avoid unnecessary re-evaluation, for example, reusing a subtree. Lastly, we describe the implementation of an incremental attribute evaluator generator by augmenting Rie (an attribute evaluator generator based on ECLR-attributed grammar) and show the results of experiments of executing the incremental attribute evaluators generated. As a result of non-optimized incremental attribute evaluation, the execution time for the re-evaluation is from 40% to 80% of the initial evaluation. In the case of optimized incremental attribute evaluation, the time for re-evaluation is from 5% to 15% of the initial evaluation. The results demonstrate the efficiency of incremental re-evaluation of one-pass attribute grammars.

1. Introduction

In this paper, we describe an incremental attribute evaluation method based on LR-attributed grammar, which is a class of one-pass attribute grammars. After the source program was parsed and evaluated, the incremental attribute evaluation for the program reevaluates only the portion affected from the modification of the source program.

One-pass AGs are quite practical for implementing one-pass compilers or the front-end of optimizing compilers. However, as far as we know, there has been no work on incremental attribute evaluation methods based on one-pass attribute grammar, although many incremental parsing methods and incremental attribute evaluation methods have been proposed for the modification of the attributed parse tree by a language-oriented editor or by an incremental parsing method.

As for incremental parsing methods, many have been developed. Celentano proposed a method for preserving configurations of LR parsing for an incremental LR parsing[Cel78]. In Jalili and Gallier's method[JG82], the parse tree is used as the internal information for the incremental parsing, and then the incremental parsing is done by dividing the parse tree into subtrees and integrating them. In [Lar95], considering the semantic analysis after parsing, the method using a parse tree and its optimal reuse is proposed. Yeh and Kastens proposed an optimal data structure for incremental LR parsing, which uses only the same number of nodes as that of tokens[YK88].

Many incremental attribute evaluation methods have also been developed: the method based on the language-oriented editor[RTD83], the one based on Ordered attributed grammar[Yeh83a], treating non-circular or circular attribute grammar[Jal85], and so on. The base attribute grammars for these methods are not a one-pass-type attribute grammar, and incremental attribute evaluation of this class of AGs has not been fully investigated.

In this paper, we show the effectiveness of combining incremental evaluation and parsing in LR-attributed grammars using a compact data structure to realize a time- and space- efficient programming environment.

First, an incremental parsing and attribute evaluation method of LR-attributed grammars, which uses an attributed parse tree, is described. In Section 2, an incremental parsing method is described. It is possible that there exist some parts that are unnecessarily reparsed during incremental parsing in practice. If the modification includes a part that is the same as the original input, that part need not be reparsed. By finding such a part and reusing the parse subtrees corresponding to it, the incremental parsing becomes more efficient. The optimization for avoiding this unnecessary reparsing is also described. In Section 3, LR-attributed grammar is explained. In Section 4, an incremental attribute evaluation method based on the incremental parsing in Section 2 is described. Also, optimization to avoid unnecessary attribute re-evaluation is mentioned.

Next, we show that the above method using an attributed parse tree is not efficient for space, since an attribute evaluator based on the LR-attributed grammar can evaluate attributes without making an attributed parse tree. Thus, we groped for a space-efficient data structure. We found that augmenting the data structure used in Yeh's incremental parsing method [Yeh83b, YK88] improves the space efficiency. In Section 5, we describe this data structure and an incremental attribute evaluation method using this data structure.

In Section 6, the implementation of the incremental attribute evaluators and the experiments are described. We implemented a Pascal-S semantic checker and a Pascal-S compiler which emits P4 code.

Section 7 presents the discussion of LR-AGs and the implementation method, and Section 8 summarizes our contribution.

2. Incremental LR Parsing

This section describes an incremental LR parsing method which is the basis of the one-pass incremental attribute evaluation to be shown later.

2.1. Overview of incremental LR parsing

Let us consider an incremental parsing of input w' after modifying w. Suppose that the original input is $w = x_0 y_1 x_1 \cdots y_n x_n$, and that the modified input is $w' = x_0 z_1 x_1 \cdots z_n x_n$. For simplicity, the case of only one modification, that is, $n = 1$, is treated in the following, without losing the ability to generalize. Only the affected portions from the modification need to be parsed if the result of the parsing of w is available. Intuitively, parsing actions for w and w' are the same from the first token to the last token of x_0. Similarly, parsing actions for w and w' are the same after a certain point where the lookahead token is in x_1. We can say this formally as follows.

A *configuration* of an LR parser is a pair whose first component is the stack contents and whose

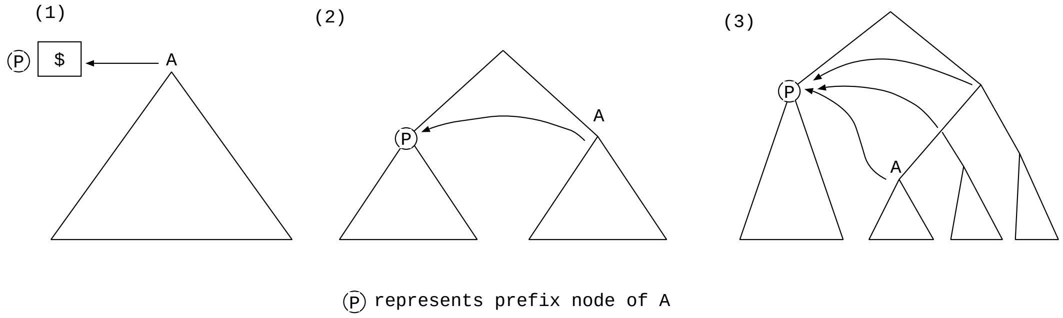

Figure 1: Examples of prefix node for A

second component is the unexpended input. A sequence of configurations shows a history of an LR parsing.

Considering the sequences of configurations for w and w', there exist i and j such that the stack part of the first i configurations are the same and the last j configurations are the same, but the configurations at the middle of either sequence are different (there may be some part in the middle of the sequences where the stack components are the same).

Therefore, if we have the history of the parsing of w, we can start parsing w' by restoring the stack of the ith configuration, and then parse the middle part of w'. We can terminate the parsing when the configuration becomes the jth-to-the-last configuration of the history.

However, this naive method is not efficient. Many data structures for incremental parsing have therefore been proposed. Generally, the parse tree is used as the model for incremental parsing.

2.2. Incremental LR parsing using a parse tree

In order to explain the incremental parsing method, first, the prefix node is defined (Fig. 1).

Definition 1 We define the prefix node of a node A ($prefix(A)$) on a parse tree as follows. We assume that there is a special node \$ for the initial state.

1. if A is the root node, then $prefix(A)$ is \$,
2. if A has the left sibling, then $prefix(A)$ is that sibling;
3. otherwise, $prefix(A)$ is the prefix node of the parent node of A. □

From the property of LR parsing, the element of $prefix(A)$ is stored directly under the element corresponding to node A in the parsing stack, i.e., stack $= (\cdots, prefix(A), A)$.

We use again $w = x_0y_1x_1$ and $w' = x_0z_1x_1$. We assume that there is the parse tree for w. An incremental LR parsing consists of two parts:

1. reusing the parse (sub)tree for x_0
2. parsing z_1 and x_1 (the parsing can be terminated when a certain condition holds)

2.2.1. Reusing the parse (sub)tree for x_0

We assume that every node of the parse tree has the LR state when it is made.

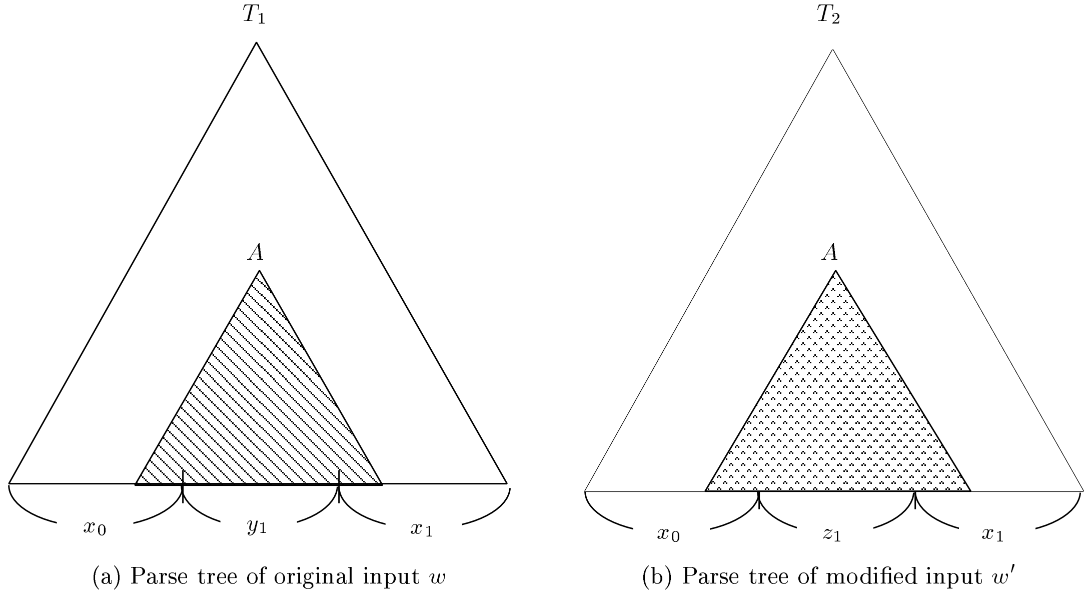

Figure 2: A model of incremental parsing (A is the *ReplacePoint*).

For the incremental parsing of w' when w is given, the actions of the parsing of w' are the same as those of w until the last token of x_0 is shifted. This is due to the fact that the next action is decided by the current lookahead and the present LR state during an LR(1) parsing.

Thus, x_0 need not be reparsed. It is possible to reconstruct the parsing stack when the last token of x_0 is shifted by visiting the prefix nodes one-by-one, starting from the last token node of x_0.

2.2.2. Parsing z_1 and x_1

Since we have restored the parsing stack when the last token of x_0 is shifted, the parsing can be started with the input z_1x_1. After parsing z_1, if a certain condition holds when the lookahead token is in x_1, parsing is not needed anymore because the parse actions are the same as in w from here on. In Fig.2, T_1 stands for the parse tree of w and T_2 stands for the parse tree of w'. Among the subtrees, only those whose root node is indicated by A are not the same, because both have different inputs y_1 and z_1.

If a node like the above A is found, the incremental parsing can be terminated and the subtree rooted at A of T_1 replaced with the subtree rooted A of T_2 becomes the parse tree of w'. In the following, we call the root node of such a subtree *ReplacePoint*.

Generally, there may be several candidates for the *ReplacePoint*. It would be most efficient if the minimum subtree can be found. An algorithm to find it is as follows:

Algorithm 1 *(find ReplacePoint)*

Check below just before a shift action in parsing w'.

1. The lookahead is in x_1, and
2. the LR state of the prefix node of the lookahead in the parse tree of w and the current LR state in parsing w' are the same, and

3. the prefix node of the prefix node of the lookahead in the parse tree of w and the prefix node of the node corresponding to the top element of the stack in the parse tree of w' are the same.

If all the conditions above hold, the prefix node of the lookahead in the parse tree of w and the node for current state in the parse tree of w' are the $ReplacePoint$. □

The proof for these conditions is written in [Sas88].

2.3. Optimization of Incremental LR Parsing Reusing Subtrees

Consider parsing of x_1. Before the $ReplacePoint$ is found, there may be some actions that are the same for the parsing of w and w'. The same actions can be skipped using the property of LR(1) parsing below.

Property 1 Assume that at some point of the LR(1) parsing, the part x_1 is analyzed, and that the configurations c and c' for w and w', respectively, are
$c = (S_0 S_1 \cdots S_n, r_1 r_2 \cdots r_N)$
$c' = (S'_0 S'_1 \cdots S'_m, r_1 r_2 \cdots r_N)$
where S_i and S'_j stands for an LR state, and r_i stands for a token of unexpended input. If $S_n = S'_m$, then the actions, until a reduction that S_n is removed from the parsing stack, are the same in the parsing of w and w'. □

This means that if the LR states are the same in w and w' for symbol X_i in production $X_0 \to X_1 \cdots X_n$, re-parsing of $X_{i+1} \cdots X_n$ can be skipped. Based on this property, we can describe an optimized algorithm for an incremental parsing as follows:

Algorithm 2 *(optimized incremental parsing)*

Assume that the parsing of w has been done and the parse tree is made. Assume also that the parsing of w' is made according to the method of section 2.2.1; that is, the parsing stack when the last token of x_0 is shifted is restored and then parsing of z_1 is performed. Now, the lookahead is a token in x_1. The symbol t stands for the lookahead.

```
i= LR state when the lookahead token is shifted in the parsing of w'
A= the node for t; /* in the parsing of w' */
j= LR state when the same lookahead token is shifted in the parsing of w
B= the node for t in the parse tree of w;
while(i==j){
  /* the condition in property 1 holds */
  /* the actions until the parent node is made by a reduction
     are the same as in the parsing of w */
  reuse the right siblings of A;
  pop elements corresponding to the left siblings of A (if any);
  i = calculate the LR state using the symbol of the parent node of B;
  push the LR state;
  make the parent node for A;
  A = the node;
  j = the LR state of parent node of B;
  B = the parent node of B;
}
```

□

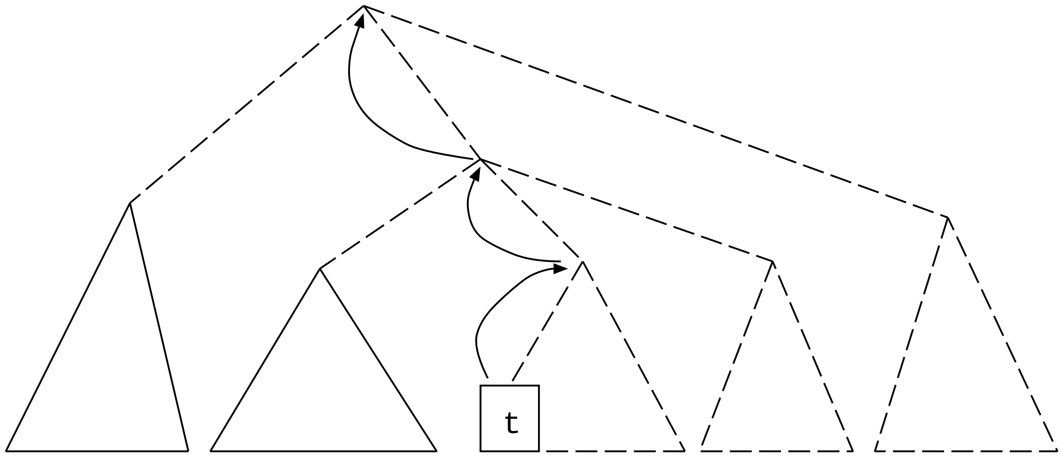

Figure 3: Image of applying algorithm 2

The detail of this algorithm is given in [NYN96].

The image of applying algorithm 2 is shown in Fig. 3. The area shown in solid lines is the part already parsed, and the area in dotted lines is the reusable part when the condition above holds.

3. LR-attributed Grammars

LR-attributed grammars (LR-AG) are a class of attribute grammars such that the attribute evaluation can be done during LR parsing. While there are several variants of LR-AGs[AMT90, Tar88, SIN85], we outline LR-AG based on [SIN85].

L-attributed grammar is the attribute grammar restricted so that every attribute associated with a grammar symbol depends only on the attributes of the symbols to the left of it in the production or the attributes of the lefthand symbol of the same production. If the underlying grammar is LL(1), one-pass attribute evaluation of L-attributed grammar can be performed during recursive descent parsing.

LR-AG should satisfy the condition that it is possible to evaluate the attributes during LR parsing. Since an LR parsing proceeds from left to right, LR-AG has to have the L-attributed property to evaluate the attributes during parsing.

In LR-AGs, it seems at a glance difficult to evaluate the inherited attributes that are passed from the ancestors during LR parsing, because the ancestors do not appear yet in LR parsing, which proceeds from bottom to top. However, in the LR parsing method, LR states are computed and we know the set of nonterminals that may have a token as the leftmost descendant. The dot (LR marker) of an LR item represents the information about the point of analysis during parsing.

For an LR state that has LR items including a dot just before the leftmost terminal symbol token in a production, there may be nonterminals which follow a dot in the same state. That is, such nonterminals have the token as the leftmost descendant. Inherited attributes of such nonterminals are calculated at that LR state based on this property. To sum up, it is possible to evaluate the inherited attributes just before a token is shifted.

The attribute evaluator based on LR-AG

A one-pass attribute evaluator based on LR-AG can be implemented by augmenting an LR parser with an attribute evaluator using attribute stacks. For example, an evaluator may have two attribute stacks, one for synthesized attributes and the other for inherited attributes. Each element of an attribute stack is a set of attributes [1]. Inherited attributes are pushed onto the stack when they are evaluated just before a token is shifted, and synthesized attributes are pushed onto the stack when they are evaluated immediately after a shift or a reduction.

In LR-AGs, attribute evaluation can be made without parse tree. But if the parse tree is made, attributes can be attached to the parse tree as follows:

1. A synthesized attribute is attached to a node corresponding to the grammar symbol.
2. An inherited attribute is attached to the current node, i.e., the node corresponding to the topmost element of the parsing stack, because an inherited attribute is evaluated just before a shift or a reduction with ε-rule. That is, the node to be attached is the prefix node of the node for the lookahead (to be shifted).

4. Incremental Attribute Evaluation Based on LR-attributed Grammar Using a Parse Tree

In this section, an incremental attribute evaluation method based on LR-AG using the attributed parse tree is presented. The method works based on the incremental parsing shown in Section 2. In the following, we use w and w' defined in Section 2. Hereafter, "initial" parsing/evaluation means parsing/evaluation of original input w.

The following should be taken into account:

1. reusing the result of the initial parsing and attribute evaluation for x_0
2. parsing and attribute evaluation of the modified part and its termination(when a certain condition holds)

4.1. Reusing the result of the initial parsing and attribute evaluation for x_0

We assume that we have the attributed parse tree for w as presented in Section 3. We can restore the attribute stack when the last token of x_0 was shifted in the same way as restoration of the parsing stack shown in Section 2.2.

4.2. Parsing and attribute evaluation of the modified part and its termination

Now, we can parse and evaluate z_1x_1 after restoration of the parsing stack and the attribute stack. If a certain condition holds, the evaluation can be terminated.

We assume that the *ReplacePoint* for parsing shown in Section 2 (Fig. 2) is already found. This means that the parse tree of w' is the same as that of w except for the subtree rooted at the *ReplacePoint*. For w', the left siblings of the *ReplacePoint* are also the same as in the parsing of

[1] For example, we can collect the attributes in a record type data structure like "struct" in C language.

w. From this and that LR-AG is assumed, the attributes of the left siblings of the *ReplacePoint* are the same. The inherited attributes of the *ReplacePoint* are also the same since the *ReplacePoint* is already found, that is, the LR state just before the grammar symbol of the *ReplacePoint* is the same as in the initial evaluation.

Since the leaves of the descendant of the *ReplacePoint* in w' are different from the leaves in the parse tree of w, the synthesized attributes of the *ReplacePoint* may have values different from those in the initial evaluation. So, they may affect evaluations thereafter; that is, they may be used to evaluate the inherited attributes of the right siblings of *ReplacePoint* and the synthesized attributes of the parent.

Thus, we should check the synthesized attributes of the *ReplacePoint*. If the synthesized attributes of the *ReplacePoint* in w' are the same as in w, they do not affect evaluations thereafter. Therefore, to terminate the incremental attribute evaluation, it is enough to add the next condition to the conditions of Algorithm 1:

(4) the synthesized attributes of the *ReplacePoint* in both the attributed parse tree of w and w' are the same.

4.3. Optimization of incremental attribute evaluation

In this section, an incremental attribute evaluation with the optimization presented in Section 2 is described.

Suppose that reduction to $X_i (1 \leq i \leq n)$ of a production "$X_0 \rightarrow X_1 X_2 \cdots X_n$" occurred and parse actions can be omitted until the reduction to X_0 as mentioned in property 1 of Section 2.3. This means that the leaves of the subtrees rooted at $X_{i+1}, \cdots, X_n$ are the same as in the parse tree of w. Let us consider the reuse of these subtrees including attributes. With LR-AG, the inherited attributes of X_{i+1} depend on the inherited attributes of X_0 and the synthesized attributes of $X_1, \cdots, X_i$. The synthesized attributes of X_{i+1} depend on the inherited attributes of X_{i+1} and the synthesized attributes of the leaves of the subtree whose root is X_{i+1}. From the above, if the inherited attributes of X_0 and the synthesized attributes of $X_1, \cdots, X_i$ are the same as those in the evaluation of w, the inherited attributes of X_{i+1} are guaranteed to be the same as those in the initial evaluation and, furthermore, the values of the synthesized attributes of X_{i+1} are guaranteed to be the same and so on; and at last, the values of the synthesized attributes of X_n are guaranteed to be the same as before. Thus we can reuse the subtree rooted at $X_{i+1} \cdots X_n$ and omit attribute evaluation for these subtrees.

Note that in this way, if n is large, this optimization does not work efficiently. However, we counted the number of symbols in the righthand side of the yacc specification for C and Pascal, and found that most productions have from 1 to 3 symbols in the righthand side. Therefore, there is no problem with using the above method.

5. Incremental Attribute Evaluation of LR-attributed Grammars Using Space-Efficient Data Structure

So far, we have described an incremental attribute evaluation using the parse tree. But, it is preferable to use a simpler data structure, for next two reasons:

1. We implemented the incremental attribute evaluator presented so far and counted the number of nodes of the parse tree. We found that the number of nodes is three times that of tokens. Considering that an ordinary attribute evaluator using a space-efficient data structure based on LR-AG can work without a parse tree, the tree data structure is not efficient in space.

2. The subtree rooted at the *ReplacePoint* in w' includes old (reused) portions and new portions, mixed. Therefore, it is difficult to manage them.

In this section, the incremental parsing algorithm by Yeh and Kastens[YK88] and its data structure are described, and we extend it for an incremental attribute evaluation based on LR-AG.

5.1. Yeh's incremental parsing

Fig.4 shows the data structure proposed by Yeh and Kastens. In their paper, they called it "Input Representation (IR)".

```
struct Node {
  char*        lexeme;
  struct Node* prev;
  int          Nonterm;
  int          no;
  struct Node* LINK;
  int          state_s;
  int          state;
};
```

Figure 4: Yeh's Data Structure

The node shown in Fig.4 represents a token. The fields of this node have the following meanings. `LINK` is a pointer to a prefix node. In Yeh's method, the information about a prefix node is set during parsing. `state` is assigned an LR state and `Nonterm` is assigned a nonterminal. `state` and `Nonterm` of the last node of IR are changed whenever a reduction occurs. `state_s` [2] has an LR state when the token is shifted. `state_s` is needed to restore the state of the token of the node; that is, if only `state` is used, `state` may have another state when a reduction occurs. `prev` points to the previous node. `no` has a serial number from the first token.

Next, we show the actions of the parser with IR. In the following, variable `LINKEND` points to the last node of IR.

Algorithm 3 *(LR parser using IR by Yeh)*

Initialization: make the node $ for initial state.

Shift:

1. make a new node for the shifted token.
2. let `prev` point to `LINKEND`.
3. let `LINK` point to `LINKEND`.
4. let `LINKEND` point to the new node.

Reduction: (k elements are popped from the parsing stack)
traverse k nodes from `LINKEND` along the `LINK`s, and let the `LINK` of `LINKEND` point to that node.

□

[2]This field is not in the original, but we introduce this to make implementation simple.

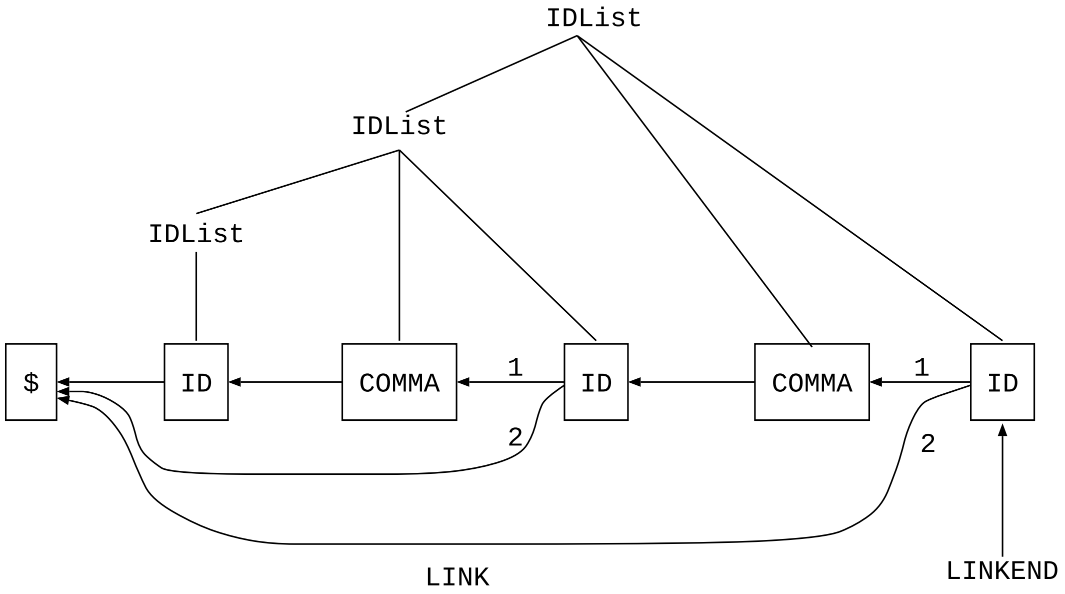

Figure 5: An example of using IR (The numbers 1 and 2 above represent the order of rewriting the LINK field.)

An example of using IR is shown in Fig.5. Suppose that example grammar includes a production "`IDList : IDList COMMA ID | ID;`", and the input is "`ID COMMA ID COMMA ID`". When the first `ID` is shifted, the node for it is made, its `LINK` points to the node of $, and its `state_s` is set. When the reduction to `IDList` occurs, the LR state is assigned to `state` of the node for `ID`. When `COMMA` is shifted, the same actions are performed. When the second `ID` is shifted, the node for it is made similarly to the first `ID`. When the reduction to `IDList` using "`IDList : IDList COMMA ID`" occurs, the `LINK` field of the node for `ID` is rewritten to point to $.

If we have the result of parsing w with IR, an incremental parsing method can be described as follows.

Algorithm 4 *(incremental parsing by Yeh)*

Reusing the result of the initial parsing for x_0: We need to restore the parsing stack when the last token of x_0 is shifted. Since the `LINK` field means the pointer to the prefix node, we can restore the parsing stack traversing the nodes from `LINKEND` along with `LINK`s.

Parsing of the modified part and termination: In the case of using IR, parsing of the modified part is performed as usual, and we can terminate the parsing if the *ReplacePoint* is found. The *ReplacePoint* can be found by algorithm 1 using `LINK`.

5.2. Incremental attribute evaluation using IR

In this section, we extend Yeh's incremental parsing algorithm to evaluate the attributes during parsing based on LR-AG. We add the following fields to Fig.4.

```
synt_type     synt;
synt_type     synt_R;
inh_type      inh;
```

`synt` records the synthesized attributes of a token when it is shifted from the lexer. `synt_R` is for storing the synthesized attributes of nonterminal A, which is the lefthand side symbol of a production when a reduction occurs. `inh` contains the inherited attributes, and the `inh` field of a node is assigned just before the next node is made, as described in Section 3.

The incremental attribute evaluation using IR is as follows:

Algorithm 5 *(incremental attribute evaluation using IR)*

Restoring the parsing stack and the attribute stacks: The restoration is done by traversing along `LINK` fields.

Attribute evaluation for the modified part and termination: The condition to terminate the incremental attribute evaluation is the same as shown in Section 4.

5.3. Optimization of incremental attribute evaluation using IR

We described an optimization of incremental LR parsing with a parse tree in Section 2.3. Algorithm 2 works well with the parse tree data structure, but we need a new algorithm for IR since there are not always nodes needed by algorithm 2. For example, when we use a right recursive production, in each IR node, only information for a node of a token, and the root node of the subtree whose rightmost descendant is the token, are stored; but information for nodes between the token node and the root node are omitted. However, algorithm 2 may need to use these nodes.

Suppose that A stands for a node for S_n in property 1, and B is the right sibling of A; that is, A is a prefix node of B. We assume the same configurations for w and w' as property 1, and we also assume that the next action is shift in parsing of w'. If $S_n = S'_m$, we can reuse the subtree rooted at B.

In [YK88], the necessary and sufficient condition for the above reuse is described as follows:

Assume the same configuration as property 1. Let the grammar symbol of the root node of a subtree B be B. Let $f(s,l)$ and $g(s,v)$ denote the action function and the goto function, respectively, in LR parsing, where $s, l, andv$ are an LR state, a lookahead, and a grammar symbol, respectively.

Let the current lookahead be r_1 and the current LR state such that the next action is shift be s. $Follow(B)$ stands for the next token of the rightmost token of the subtree rooted at B.

The necessary and sufficient condition given in [YK88] is

$$f(g(s,B), Follow(B)) \neq error.$$

If this condition holds in the incremental parsing of w' for the parsing of w, the parse actions which make the subtree rooted at B are the same in both parsing of w and w', and optimization which reuses subtrees becomes possible.

However, the condition above only decides whether the subtree rooted at B can be reused. Because we do not have information (that is, pointer to a node in the right) to find such a subtree if we use IR, we have to extend the IR to find B. Therefore, we add a field `RLINK` to the IR node. `RLINK` of a node A is set to point to B when the `LINK` of a node B is set to point to A. Now, if the condition above holds, we can find the reusable subtree B by traversing along `RLINK`.

Now, we can reuse subtrees if only parsing is concerned, but we have to consider the condition of reusing subtrees including their attributes.

Since a reusable subtree decided by the condition above is included in the part x_1 and does not have new leaves, i.e., the modified portion, the attribute evaluation inside the subtree is affected by its inherited attributes only. Then the condition to reuse the subtree including its attributes is that its old inherited attributes and current input attributes are the same. We can get the old input attributes from `LINKEND`.

However, since we do not have synthesized attributes for the root of the reusable subtree, that is, since the node of B pointed by `RLINK` of A has `Nonterm` which may not be the same as B, the node has only synthesized attributes of the `Nonterm`, we have to store them somewhere. So, we add a field for them to IR, and the synthesized attributes of the root of the reusable subtree are stored in the node whose `RLINK` points to the root of the reusable subtree.

Thus, we get an optimized incremental attribute evaluation algorithm using IR.

6. Implementation and Experiments

In this section, we describe implementations of incremental attribute evaluation methods presented above and some experiments to measure their efficiency.

6.1. Implementation

We have augmented the attribute evaluator generator Rie [SIN95] based on LR-AG to make an incremental attribute evaluator from a specification automatically. Since Rie is made based on Bison [DS91], Rie emits the attribute evaluator together with the LR parser.

The augmented Rie emits the program which includes `yyparse` for the initial attribute evaluation, and `yyincparse` for the incremental attribute evaluation. We made three augmented versions of Rie for experiments:

A perform non-optimized evaluation using parse tree

B perform optimized evaluation using parse tree

C perform optimized evaluation with IR

6.2. Experiments

The objective of the experiments was to demonstrate the efficiency of incremental attribute evaluation methods. Especially, we wanted to know the efficiency of the optimizations. We made three versions of a semantic checker of Pascal-S [Ber81] and of a compiler of Pascal-S which generates Pascal-P4 code [PD82] by the augmented Rie above. We measured the running times using the "pixie" command on Indy (CPU:R4600 100MHz) of SGI. Pixie reports machine cycles of the execution of a program.

We measured the running times of the parsing and the attribute evaluation of the following modification:

1. inserting a `writeln;` in a quicksort program.

2. changing `repeat` statement to `while` statement in a quicksort program.

3. changing `while` statement to `repeat` statement in a program for calculating π.

1. is a case of simple modification. 2. and 3. is a case in which the instruction for repetition is modified but the including statements are unchanged.

The result is shown in Table 1. A, B and C denote three versions described in the previous section. For comparison, D denotes a non-incremental attribute evaluator produced by Rie that uses the same

	initial evaluation				re-evaluation			
	A	B	C	D	A	B	C	D
1	841,657	887,683	887,057	396,098	318,418 (38%)	16,575 (2%)	19,657 (2%)	396,898
2	838,626	887,064	885,970	396,098	323,273 (39%)	42,068 (5%)	87,098 (10%)	397,550
3	1,849,006	1,953,489	1,931,072	910,875	728,126 (39%)	61,004 (3%)	102,311 (5%)	910,665

unit:1000machine cycles

Table 1: The time of incremental attribute evaluation

lexer as others and does not create nodes. The figures in parentheses, e.g. (38%), show the time ratio of re-evaluation to the initial evaluation.

From the result, we see that for all cases the re-evaluation time is shorter than the initial evaluation. For example, the result of B for 1 shows re-evaluation takes only 1/50 of its initial evaluation. As for optimized versus non-optimized re-evaluation, the time of the optimized re-evaluation (B, C) is about 1/20 to 1/4 of the non-optimized one (A).

On the other hand, comparing optimized evaluation using a parse tree (B) and using IR (C), the former is mostly a little faster in this experiment.

Comparing with non-incremental evaluator, each evaluation time of A, B and C is roughly double of non-incremental one (D) for initial evaluation. As for re-evaluation, the execution time of non-optimized one (A) is almost the same as non-incremental one (D), but the execution time of optimized ones (B or C) is from 4% to 22% of non-incremental one.

The next experiment is for a practical compiler, a Pascal-S compiler emitting P4 code. Because the method to emit P4 code is by synthesizing the code of each subtree and making a new code, the condition to terminate the incremental attribute evaluation does not hold until acceptance of the whole input. Of course, the optimization described in Section 4 and 5 is applied during the evaluation. The result of this experiment is shown in Table 2. We measured only the attribute evaluation time, not including the time of I/O which emits code to output file. The cases 1, 2 and 3 are the same as in Table 1.

	initial evaluation				re-evaluation			
	A	B	C	D	A	B	C	D
1	1,281,757	1,347,520	1,348,176	625,425	1,037,838 (81%)	104,698 (8%)	95,950 (7%)	627,865
2	1,280,944	1,347,192	1,347,417	625,425	1,034,742 (81%)	144,698 (11%)	197,568 (15%)	628,118
3	2,847,447	2,987,580	2,979,086	1,455,444	1,081,747 (38%)	139,341 (5%)	180,413 (6%)	1,453,853

unit:1000machine cycles

Table 2: The time of incremental attribute evaluation of Pascal-S compiler

In Table 2, the re-evaluation using IR (C) is faster than that using the parse tree (B) in case 1 but it is not in case 2 or 3.

These results show that as for time efficiency the optimization using IR works well in some cases, and in other cases the optimization using the parse tree works better.

As for the comparison with non-incremental evaluation, non-optimized re-evaluation (A) is 1.6 times slower than non-incremental one (D). Optimized re-evaluation (B or C) runs with from 10% to 30% time of non-incremental one (D).

As described above, since the condition to terminate the incremental attribute evaluation of Pascal-S compiler does not hold until acceptance of the whole input, the efficiency of the re-evaluation is a little worse than the incremental attribute evaluator of Pascal-S semantic checker. In any case, optimizations that skip re-evaluations are important for an incremental re-evaluation.

In order to measure the space requirement, we counted the number of tokens and nodes created during the optimized incremental attribute evaluation using parse tree. The result is shown in Table

3. The case 1, 2 and 3 are the same as before. The result shows that in every case the number of nodes is about 3 times that of tokens.

In our implementation, the size of a node of the parse tree by the method using the parse tree is 80 bytes. The size of a node of IR is 100 bytes.

So, if a program contains x tokens, the method using parse tree requires $3x \times 80$ bytes. Recalling that the nodes used in IR are just tokens, the method using IR requires $100x$ bytes. Then, the memory requirement of the method using IR is only 40% of the other.

To sum up, as for time, the method using parse tree and using IR work with similar efficiency, and as for space efficiency, the method using IR is better than the other.

	initial evaluation		re-evaluation		total	
	tokens	nodes	tokens	nodes	tokens	nodes
1	281	927	2	7	283	934
2	283	930	11	35	294	965
3	593	1960	17	52	610	2012

Table 3: The number of tokens and nodes created using incremental attribute evaluation algorithm with parse tree.

7. Discussion

7.1. LR-AGs

The LR-AGs are of course a limited class of attribute grammars. However, in practice new languages and their processors are developed one after another around the world. Not all of them are large-scale, rather many of them are middle- or small-scale languages. For programming educations, C, Pascal or their subset languages are widely used. Most of those languages can be processed in a single pass, if we do not care optimization. Therefore we think application of incremental evaluation of LR-AGs to semantic checkers or non-optimizing compilers is useful, such as those presented in Section 6. Furthermore, the front end for almost all languages can be realized with one-pass AGs. In fact, our group use LR-AG in the front-end and another attribute evaluator based on a circular remote attribute grammar [SS00] in the back-end to implement optimizing compilers. One-pass AGs have the advantage of good performance in their attribute evaluators.

7.2. Implementation method

We use the C language for implementation since we use Rie. On the other hand, according to the formalism of attribute grammars, it may be proper in some cases to use a functional programming language for the implementation of an attribute evaluator. There are investigations of incremental attribute evaluation using the technique of deforestation, and an implementation has been done for HAG [SS99]. It is not easy to compare the implementation in C and in a functional language by a single criterion. However, from the experience of actually implementing in C, we think the method described in this paper would be accepted on the line of traditional compiler construction.

8. Conclusions

In this paper, incremental attribute evaluation methods of LR-attributed grammars with parse trees and with space-efficient data structures have been described. In order to improve space efficiency, the data structure proposed by Yeh et al. has been augmented for incremental attribute evaluation. Optimizations of re-evaluation were also presented. As a result of implementing the incremental attribute evaluation methods and the experiments, we have shown that the optimization to avoid unnecessary attribute re-evaluation is quite important.

Bibliography

[AMT90] Rieks op den Akker, Bořivoj Melichar, and Jorma Tarhio.
The Hierarchy of LR-Attributed Grammars.
In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lect. Notes Comput. Sci.*, pages 13–28. Springer-Verlag, Oct 1990.

[Ber81] R.E. Berry.
Programming Language Translation.
Ellis Horwood Limited, 1981.

[Cel78] A. Celentano.
Incremental LR Parsers.
Acta Inf., 10:307–321, 1978.

[DS91] C. Donnelly and R. Stallman.
Bison Reference Manual, 1991.

[Jal85] Fahimeh Jalili.
A General Incremental Evaluator for Attribute Grammars.
In *Science of Computer Programming*, volume 5, pages 83–96, 1985.

[JG82] Fahimeh Jalili and Jean H. Gallier.
Building Friendly Parsers.
In *ACM Ninth Symposium on the Principles of Programming Languages*, pages 196–206, 1982.

[Lar95] J.-M. Larchevêque.
Optimal Incremental Parsing.
ACM Trans. Prog. Lang. Syst., 17(1):1–15, January 1995.

[NYN96] H. Nakai, Y. Yamashita, and I. Nakata.
An Incremental LR Parsing Method and its Evaluation.
Trans. IPSJ, 37(3):371–383, 1996.
(in Japanese).

[PD82] Steven Pemberton and Martin Daniels.
Pascal Implementation – The P4 Compiler.
Ellis Horwood Limited, 1982.

[RTD83] Thomas Reps, Tim Teitelbaum, and Alan Demers.
Incremental context-dependent analysis for language-based editors.
ACM Trans. Prog. Lang. Syst., 5(3):449–477, July 1983.

[Sas88] M. Sassa.
Incremental Attribute Evaluation and Parsing Based on
ECLR-attributed Grammar.
Tech. Report ISE-TR-88-86, Inst. of Inf. Science, Univ. of
Tsukuba, 1988.

[SIN85] M. Sassa, H. Ishizuka, and I. Nakata.
A Contribution to LR-attributed Grammars.
J. Inf. Process., 8(3):196–206, 1985.

[SIN95] Masataka Sassa, Harushi Ishizuka, and Ikuo Nakata.
Rie, a compiler generator based on a one-pass-type attribute grammar.
Software – Practice and Experience, 25(3):229–250, 1995.

[SS99] João Saraiva and Doaitse Swierstra.
Data structure free compilation.
In *CC: International Conference on Compiler Construction, LNCS*,
1999.

[SS00] Akira Sasaki and Masataka Sassa.
Circular attribute grammars with remote attribute references.
In *International Workshop on Attribute Grammars and their Applications (WAGA '00)*, Ponte de Lima, Portugal, July 2000.

[Tar88] Jorma Tarhio.
A Compiler Generator for Attribute Evaluation During LR
Parsing.
In Dieter Hammer, editor, *Compiler Compilers and High Speed Compilation*, volume 371 of *Lect. Notes Comput. Sci.*, pages 146–159.
Springer-Verlag, Oct 1988.

[Yeh83a] Dashing Yeh.
On Incremental Evaluation of Ordered Attributed Grammars.
BIT, 23:308–320, 1983.

[Yeh83b] Dashing Yeh.
On Incremental shift–reduce parsing.
BIT, 23:36–48, 1983.

[YK88] Dashing Yeh and Uwe Kastens.
Automatic Construction of Incremental LR(1)-Parsers.
SIGPLAN Notices, 23(3):33–42, 1988.