ISE-TR-88-66

INCREMENTAL ATTRIBUTE EVALUATION AND PARSING BASED ON ECLR-ATTRIBUTED GRAMMARS

by

Masataka SASSA

March 19, 1988

INSTITUTE
OF
INFORMATION SCIENCES AND ELECTRONICS

UNIVERSITY OF TSUKUBA

Incremental Attribute Evaluation and Parsing Based on ECLR-attributed Grammars

Masataka Sassa

Institute of Information Sciences and Electronics
University of Tsukuba
Tsukuba-shi Ibaraki-ken 305 Japan

34 pages
March 19, 1988

Abstract
A method of incremental attribute evaluation and parsing is described. It is based on a class of one-pass attribute grammars called ELCR-attributed grammars which works with LR parsing. The method unifies incremental attribute evaluation and incremental parsing in a single algorithm. It is expected to be space efficient with respect to inherited attributes. Multiple substitutions in the original input are also allowed.

CR Categories and Subject Descriptors:
D.3.4 [Programming Languages]: Processors - Translator writing systems and compiler generators
D.3.1 [Programming Languages]: Formal Definitions and Theory - Semantics, Syntax
D.2.3 [Software Engineering]: Coding - Program editors
D.2.6 [Software Engineering]: Programming Environments - Interactive

General Terms:
Languages

Additional Keywords and Phrases:
incremental attribute evaluation, one-pass attribute grammars, ECLR-attributed grammars, incremental parsing, LR parsing.

This report was written while the author was visiting Department of Computer Science, University of Helsinki, Finland.
The author would like to thank Ministry of Education, Japanese Government for financial support.
The report is also published as Report A-1988-9 from Department of Computer Science, University of Helsinki.

1. Introduction

The importance of interactive environments which support software developments has been highly recognized. As a typical example, let us think of an environment where a language-based editor, interpreter, debugger and code generator are unified around a single intermediate representation, as follows.

```
source  ---- language-  ---- intermediate   ---- interpreter
             based           representation ---- debugger
             editor                         ---- code generator
```

If we regard such a system as a language processor, the front-end, which backs up the editor, deals with the conversion from the source program into the intermediate representation, i.e. lexical, syntactic and (static) semantic analysis. According to the interactive nature of modification of the source program by the editor, it will be nice if the analysis is made in an incremental way.

Several systems exist so far which make incremental syntax and semantic analysis. As for incremental syntax analysis or incremental parsing, some systems allow only modification of the parse tree itself [Notkin 85]. But, recent experience with language-based editors shows that a hybrid approach which also accepts text mode editing in addition to structure mode editing is indispensable. In this sense, incremental parsing [Ghezzi 80, Jalili 82, Agrawal 83, Yeh 88] is effective.

As for semantic analysis, the use of attribute grammar [Knuth 68] is becoming popular due to its good balance between formality and easiness of automatic generation of attribute evaluators. Therefore, henceforth we adopt attribute grammars as the base and use an attributed parse tree as the intermediate representation. As for incremental attribute evaluators, previous works were mostly based on elaborate approaches which are separate from parsing [Yeh 83b] or rather expensive [Reps 83]. However, the experience in the HLP84 system [Koskimies 88] and in our Rie system [Ishizuka 85] [Sassa 85a] showed that the use of one-pass attribute grammars is efficient and practical enough.

Considering the above facts, we present in this report a unified method which performs both parsing and attribute evaluation in an incremental way in one pass. It is based on a class of one-pass attribute grammars called ECLR-attributed grammars [Sassa 87]. It works with LR parsing.

Our basic hypothesis is that we maintain the attributed parse tree (hereafter APT). This will be justified in a programming system which unifies an interactive interpreter, debugger etc. in addition to a language-based editor.

One of the main advantages of our method is that the storage for APT is space efficient due to the concept of LR-attributed grammars and equivalence classes in ECLR-attributed grammars. In particular inherited attributes must be stored only in a part of the nodes of the APT, not in every node, and inherited attributes having the same value can share a memory space. Typical storage reduction of 1/3 - 1/10 is expected for inherited attributes. Synthesized attributes are stored in each node as usual.

Our incremental parsing method is a combination of the methods of Ghezzi and Mandrioli [Ghezzi 80] and of Yeh and Kastens [Yeh 88], both for LR grammars. The former method uses a parse tree, but it is for LR(0) grammars without ε-productions (productions where the right-hand side is empty) and deals only with a single modification in the original input. The latter method is for LR(1) grammars with ε-productions and allows multiple modifications. But it uses a special data structure for space efficiency and keeps LR states in it.

Our incremental parsing method is for LR(1) grammars with ε–productions and allows multiple modifications. We use the general (attributed) parse tree as the internal structure. We need not store LR states in the APT in contrast with the methods of [Jalili 82, Agrawal 83, Yeh 88] etc. (although some uses different data structures). Elimination of LR states will be convenient in editors allowing also structure mode editing, like "cut and paste" of subtrees.

In the following, we explain incremental parsing in section 2, and incremental evaluation in section 3.

2. Incremental parsing

We assume that readers are familiar with basic concepts of grammars and LR parsing. Unless otherwise stated, the definitions and notations of [Aho 86] are used in this report.

Let G = (N, T, P, S) be an augmented LR(k) (henceforth, simply LR) grammar whose first production is of the form "$S \rightarrow S'\ \$^k$".

Suppose that $w = x_0\, y_1\, x_1\, y_2\, x_2\, ...y_m\, x_m$ is in L(G), and that w has been parsed by an LR parser, yielding the parse tree shown in Fig. 1(a).

Suppose also that $w' = x_0\, y_1'\, x_1\, y_2'\, x_2\, ...y_m'\, x_m$ is in L(G) and w' is obtained from w by substituting y'_i for y_i (i=1,...,m). (Note that x_0 or x_m may be ε, but not x_i (i=1,...,m-1). y_i or y'_i, but not both, may be ε. The last terminal of x_m is $.)

After modification, only a part of the parse tree remains "valid". By "valid", we mean that the grammar symbol labeling a node of the part and the production applied at the node are the same as in the original parse tree. In fact, only the shaded area in Fig. 1(b) is valid after modification. The zig-zag of a border in Fig. 1(b) means that there may be some productions for which some sons are in the shaded part but others are not, like the production "$A \rightarrow X\ Y\ Z$". The border in zig-zag can not be known in advance.

Let us divide x_i into three parts, i.e. $x_i = t_i\, u_i\, v_i$ (i=0, ...,m). The invalidity of the part above v_{i-1} (i =1 ,...,m) is due to the fact that lookahead symbols which involve the first part of y'_i may affect the move of the LR parser in v_{i-1}. For LR(k) parsers, it is clear that letting the length of v_{i-1} $|v_{i-1}|$ = k-1 is enough for safety. (At the boundary, v_m = ε.) (Letting $|v_{i-1}|$ be not k but k-1 comes from the fact that when the parser made a shift operation for the last symbol of v_{i-1} the k lookahead symbols were still in x_{i-1}. So the valid part of the parse tree also includes the leaf node corresponding to the last symbol of v_{i-1}. cf. section 2.2.)

The invalidity of the part above t_i is due to the possibility that the parsing configuration of the LR parser at the end of the part of y'_i might not be generally the same as when it parsed the original input. Again, the length of t_i or the border above the end of t_i can not be known in advance. (t_0 = ε. t_i may be ε.)

In Fig. 1(c)(d), a couple of other possibilities are illustrated. The shaded parts may be disconnected from each other (Fig.(c)) or several modifications may cause invalid parts to fuse into one (Fig.(d)).

To be more precise for later explanations, let $\mathrm{TAIL}_j(z)$ be the last j terminal symbols of z in w or w'. If $|z| \leq j$, this denotes the sequence of terminal symbols starting from j-th position before the end of z in w or w' (or the first terminal of w or w' if it exceeds the beginning) up to the end of z. Then, v_i means $\mathrm{TAIL}_{k-1}(x_i)$.

2.1 Outline of the incremental parser

The outline of reconstruction of the parse tree is generally as follows (Fig. 2).

First, initialize i to 1.

Then, the incremental parser recovers its parsing configuration of the moment just before reading v_{i-1} (Fig. 2(c)(d)).

Next, it parses the part $v_{i-1}\ y_i'\ t_i$ and newly makes a fragment of the parse tree corresponding to that part (Fig. 2(c)). We call it a *new parse subtree* . (It is not really a subtree because of the zig-zag in the border, but we call it as such for simplicity of terminology). Here, we generally preserve the original parse tree, preferably as much as possible. Since it is not generally possible to know the end of t_i beforehand, the incremental parser checks the *matching condition* after entering the analysis of part x_i. When this condition holds, that is, when the parsing configuration becomes the same as when it parsed the original input, the incremental parser for this part stops. (Sometimes the matching condition may not hold in the part x_i, and analysis may proceed to the following part, like y'_{i+1} etc. (Fig. 1(d)). But let us assume for the moment that the matching condition holds in the part x_i, before v_i . The precise treatment will be given in section 2.4).

Then, the new parse subtree corresponding to " ... $v_{i-1}\ y'_i\ t_i$ " is connected to the appropriate node of the original parse tree. This connection of the subtree after succeeding in reparsing is safer than modifying the original parse tree itself, in the case when syntax (and semantics) errors occur in the modified part of input, since the original parse tree will still be retained.

If there are multiple modifications, skip parsing u_i and increment i by 1.

Now, we arrive at the same situation as we started the incremental parsing for y_i'. Repeat the above steps until we reach x_m.

Note: In making a new parse subtree, we assume in the usual way that a new internal node is created at the moment of "reduce" operation and a new leaf node is created at the moment of "shift" operation. The zig-zag in the left border of the new parse subtree arises when the parser does a "reduce" operation in which some "sons" are in the original parse tree. So, the shape of the zig-zag can only be determined when we reparse the modified part.

Now, we present the incremental parser using the following grammar as a running example.

G1:
(0) $E' \rightarrow E$
(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow i$

The LR states are given in Fig. 3(a) [Aho 86]. Here, we give only the canonical collection of LR(0) items, or the core part of LR items, for simplicity, but it does not affect the generality of discussion. The parsing table is given in Fig. 4 [Aho 86].

Suppose that the original input w is

$$w = x_0\, y_1\, x_1\, y_2\, x_2 = i * i + i * i + i$$

where $x_0 = i\,*$, $y_1 = \varepsilon$, $x_1 = i + i$, $y_2 = \varepsilon$, $x_2 = *\ i + i$. The corresponding parse tree is shown in Fig. 5(a). Subscripts like i_1, T_1 are used only to discriminate occurrences in the following explanations. (Henceforth we often use subscripts and superscripts for discrimination/explanation. Their meaning will be clear.)

If we replace the part $y_1 = \varepsilon$, $y_2 = \varepsilon$ by $y'_1 = ($, $y'_2 =)$, the modified input w' becomes

$$w' = x_0\, y'_1\, x_1\, y'_2\, x_2 = i * (i + i) * i + i$$

The new parse tree is shown in Fig. 6(a). Only the shaded part of the original parse tree in Fig. 5(a) turns out to be valid after modification. In this example modification, the two invalid parts above y_1' and y_2' of Fig. 6(a) have fused into one (cf. Fig. 1(d)).

About data structures of the parse tree or APT

Although the algorithm presented here does not rely on any concrete data structure, it might be helpful if an example data structure is given. Readers interested in an example data structure are referred to Appendix 1.

2.2 Initialization of the incremental parser

In order to initialize the incremental parser, let us introduce some concepts .

For each node *n* in the parse tree, let *prefix* (*n*) be a function or a field of *n* which gives a pointer to either (a) its left brother node (if one exists), or (b) the left brother node of the closest ancestor that has a left brother (if such an ancestor exists), or (c) nil (otherwise). (This corresponds to the rightmost thread of [Ghezzi 79] or LINK field of [Yeh 88]).

For a node *n*, let us consider a sequence of nodes by succesive application of prefix() starting from *n*. Assume that *n* is the beginning of the sequence and the node immediately before "nil" is the end of this sequence. Let us call it *prefix chain*. It actually corresponds to the reverse of the viable prefix in LR parsing [Aho 86]. As an example,the prefix chain of $*_2$ in Fig. 5 is

$$*_2 \ T_1$$

and that of i_5 in Fig. 5 is

$$i_5 \ +_4 \ E_1$$

The prefix chain of a node *n* can be easily got as follows.

```
function Trace prefix chain(n):
current_node :=n ;
prefix_chain := ε ;
loop
   append current_node to prefix_chain ;
   while current_node is the leftmost son of a node do
         current_node := current_node's father ;
         if current_node = root node then return ( prefix_chain ) ;
   end while ;
   current_node := current_node's left brother ;
end loop ;
```

Now, consider the initialization of the incremental parser for part y'_i. We assume that

$$\dots y'_1 \dots y'_2 \dots \dots y'_{i-1}$$

have been already parsed and their parse subtrees are connected to the original parse tree. (#)

The incremental parser skips parsing of the shaded part above u_{j-1} and sets up its parser configuration at the moment when it has just shifted the last terminal symbol of u_{i-1} (Fig. 2(c)(d)). The initialization can be done by using the prefix chain

$a, X_{n-1}, X_{n-2}, \dots, X_1$

starting from the last terminal symbol a of u_{i-1}. (In case of LR(1) grammars, a is in fact the last terminal symbol of x_{i-1} itself.) (If $i-1 = 0$ and $|x_0| \le k-1$, let a be the first terminal symbol of x_0.) It is known that for any teminal symbol a, the configuration of the parse stack at the moment where a has been shifted can be obtained by this prefix chain [Yeh 88]. If we assume (#), the subtree to the left of this prefix chain is assured to be valid after the modification.

Note regarding ε-productions:

Reductions by ε-productions might have occured in the original parse tree. Especially, there may be several ε's immediately before or after a, which have been reduced to some nonterminals. Here we should note the distinction between terminal symbols in the input and leaf nodes of the parse tree. ε's belong only to leaf nodes. This can be illustrated as follows:

```
     X Y Z   U V   W
   | | | | | | | | | |
   d ε ε ε a ε ε f ε g    (leaf nodes)
   d       a     f   g    (input)
```

In this case, let u_{i-1} be leaf nodes "... $d\, \varepsilon\, \varepsilon\, \varepsilon\, a$ " and v_{i-1} be the input "f g ...". a is the last symbol of u_{i-1}.

Thus, the initialization of the parser configuration, which is (parse stack, remaining input), is made as follows:

procedure Initialize incremenal parser (for y'_i):

(1) Put (nil, nil) and then the initial LR state I_0 on the bottom of the parse stack (see note 1).

(2) Get the prefix chain $a, X_{n-1}, X_{n-2}, \dots, X_1$ starting from the last terminal symbol a of u_{i-1}.

(3) If the prefix chain = ε, then skip this step.

Otherwise, put into the parse stack each grammar symbol in the above prefix chain in revese order like $X_1, X_2, \dots, X_{n-1}, a,$ recovering the corresponding LR states $I_1, I_2, \dots, I_{n-1}, I_n$ by performing LR parsing using the goto function of the parsing table. In the parse stack elements for grammar symbols, we also make a field where pointers $p_{X1}, p_{X2}, \dots, p_a$ to nodes corresponding to $X_1, X_2, \dots, a$ in the original parse tree are stored. Thus in general, the parse stack is like (note 2) (nil,nil) I_0 (X_1, p_{X1}) I_1 (X_2, p_{X2}) I_2 ... (X_{n-1}, p_{Xn-1}) I_{n-1} (a, p_a) I_n

(4) Let the remaining input be

b ... $\$^k$

where b is the input symbol next to a and $\$^k$ is the end of input.

Note 1: I_0 is the initial state including the LR item $[S \rightarrow . S' \$^k]$ where "$S \rightarrow S' \$^k$" is the first production in the augmented grammar.

Note 2: In practice, grammar symbols X_i's need not be stored [Aho 86].

Example Let us see the incremental parsing of $y'_1 = (_3$, in Fig. 6. Now, the last terminal symbol of u_0 or a in the above procedure is $*_2$. Then, the prefix chain is $*_2$, T_1. So, the parse stack will be initialized as

(nil, nil) I_0 (T_1, p_{T1}) I_2 $(*_2, p_{*2})$ I_7

where p_{T1} and p_{*2} are pointers to nodes for T_1 and $*_2$, respectively.

The remaining input is $(_3$, i_4, ... \$.

2.3 Termination of the incremental parser

After finishing the parsing of y_i' and entering x_i, the incremental parser can stop parsing when a condition that the parser is in the same configuration as it parsed the original input holds. This condition is called the *matching condition*. Actually, t_i is defined to be the part of the input from the beginning of x_i up to the position of the input where the matching condition holds (Fig. 1(b)).

Suppose that a reduction "$A \rightarrow \alpha$" occurs. Informally, if there is the same nonterminal A in the original parse tree such that the configuration of the parser when it was originally recognized is the same as the current one, we can say that the matching condition holds (Fig. 2 (a)(b)(c)(e)).

To be more precise, recall that a parser configuration is determined by

(parse stack, remaining input)

So, if the content of the "parse stack" and "remaining input" (here we only think of the input in the part x_i except v_i) are the same for the original and the current one, we can say that future moves of the parser (for the part x_i except v_i) will also be the same, due to the nature of LR parsing. (Considering the case of Fig. 1(d), the "remaining input" may be in practice the part x_j except v_j for some $j \geq i$).

Firstly, checking equality of the remaining input is trivial. If the parser is reading some part in x_i (except v_i), the remaining input is of course the same.

Secondly, to check equality between the parse stack corresponding to the original parse tree and the current parse stack, we do not need to look at all parse stack elements. It will be shown that if we have the original parse tree, it is only required to check the topmost and the next element of the current parse stack.

To check the matching condition, let *ancest(n)*, where *n* is a leaf (terminal) node in the parse tree, be a function or a field of *n* which gives the topmost ancestor that has *n* as the rightmost descendant (if such an ancestor exists), or "nil" (otherwise). (This corresponds to LAB in [Yeh 83a]). For example in Fig. 5,

$$\text{ancest}(i_5) = T_3, \quad \text{ancest}(*_6) = \text{nil}, \quad \text{ancest}(i_7) = E_2$$

Using this, the matching condition can be stated as follows.

Matching condition: (Fig. 2)

Suppose that a reduction by "$A \rightarrow \alpha$" has occurred. Let the current configuration be

$$(\ldots (X, p_X)\ l_{q-1}\ (A, p'_A)\ l_q\ ,\ d \ldots \$\ )$$

where the first component is the parse stack and the second component is the remaining input.

Note that p'_A points to a node of the new parse subtree because we have just made a reduction and p_X might be nil if it is the bottom element of the stack.

Let the terminal symbol just before *d* in the original parse tree be *c* (note E2, appendix 3). Let *n* be the node specified by ancest(*c*) in the original parse tree (note 3). The matching condition holds if

(i) *d* is in x_j except v_j for some $j \geqq i$,

(ii) " the grammar symbol corresponding to *n* " = *A*, and

(iii) prefix(*n*) = p_X (comparison of pointers, note 1).

The matching condition is assured to eventually hold, since at least it holds when a reduction to the start symbol occurs at the end of input, where $d = \$$, *n* is the root node, *A* is the start symbol and p_X = nil.

The proof is given in Appendix 2. Notes regarding ε-productions are given in Appendix 3.

Notes:

1. Condition (iii) means that the node pointed by prefix(*n*) is the same as the node pointed from the 2nd (to the top) element of the parse stack. Since *n* is in the original parse tree, prefix(*n*) points to a node in the original parse tree (or nil). Thus, both prefix(*n*) and p_X point to the same node in the original parse tree (or nil). For example in Fig. 2 (a)(c), they

both point to n_X.

2. The case of boundary conditions, e. g. $d = \$$, will be clear.

3. The matching condition can be modified by changing the definition of ancest(m). We could define ancest(m) to be "every" ancestor n of m which has m as the rightmost descendant, but not restricted to the "topmost" ancestor. Accordingly, we could check every such n of ancest(c) in the matching condition. But practically, if left recursion is mostly used in productions, the improvement seems to be small [Yeh 88].

Example See Fig. 5 and 6. Suppose that i = 1 and the incremental parser has read $i_{9'}$ and the lookahead is $+_8$. Thus, $c = i_7$ and $d = +_8$. Suppose that a reduction "E → T" occurred and the parse stack is now

$$(\text{nil,nil}) \ I_0 \ (E_2, p_{E2}) \ I_1$$

(This is exactly what will happen when we reparse the input of Fig. 6.)

The matching condition holds for this reduction "E → T" because n, which is the node specified by ancest(i_7), is the node for E_2, and

(i) $+_8$ is in X_j for $j = 2 \geq i = 1$

(ii) "the grammar symbol corresponding to n " = E, holds

(iii) prefix(n) is nil. (X, p_X) is (nil,nil), thus p_X is nil. Thus, prefix(n) = p_X, holds.

2.4 Incremental parser

We can now present the incremental parser as a whole, which is stated in the following algorithm.

Algorithm Incremental parser

Input: The parse tree of $w = x_0\, y_1\, x_1\, y_2\, x_2 \ldots y_m\, x_m$ and still unparsed input $w' = x_0\, y_1'\, x_1\, y_2'\, x_2 \ldots y_m'\, x_m$.

Output: The parse tree of w' if w' belongs to L(G), otherwise an error indication.

Method: It consists of the following steps:

(1) Set i = 1.

(2) Skip parsing of u_{i-1}. By using the procedure "Initialize incremental parser" presented before, set the parse stack to have the same contents as when it has just shifted the last terminal symbol of u_{i-1}.

(3) In the following steps (4) through (7), if "accept" or "error" turns up, go to step (8).

(4) Using the normal parser, parse the rest of v_{i-1} and y'_i while making a new parse subtree.

(5) After the lookahead is within x_i, continue parsing and making the new

parse subtree, but test the matching condition every time a reduction is made.

(6) If the matching condition does not hold yet, but the lookahead comes to be within v_i ($i < m$), increment i by one, and go to step (4).

(7) When the matching condition holds after reading t_i and at node n_A of the original parse tree, then replace the subtree of n_A by the new parse subtree for "... $v_{i-1}\ y'_i\ t_i$". Increment i by one. If $i \le m$, go to step (2).

(8) Stop.

Notes:

1. In all steps, ε-productions should be treated as mentioned before.
2. The normal parser to be used to make the original parse tree will be trivial. It will be shown later with attribute evaluation.
3. Optimization for reducing the checks of the matching condition [Yeh 88] may be adopted.

Example: When we modify the original input $w = x_0\ y_1\ x_1\ y_2 x_2$ = i * i + i * i + i to $w' = x_0\ y'_1\ x_1\ y'_2\ x_2$ = i * (i + i) * i + i , where x_0 = i * , y_1 = ε , x_1 = i + i , y_2 = ε , x_2 = * i + i , y'_1 = (and y'_2 =) , the modified parse tree and incremental parsing for the modified input are as shown in Fig. 6 and 7, respectively. Matching condition does not hold within x_1, and we go from step (6) to step (4) again. The matching condition holds at x_2 at E_2 of Fig. 6 or at E in the last line of Fig. 7. At step (7) of the algorithm, we connect E_2, which is the root of the new parse subtree, to E_3.

2.5 Discussion

Skipping the parsing of a part of t_i during the incremental parsing will be possible as mentioned in [Celentano 78, Yeh 88]. We have not included it here, in order not to complicate too much the incremental attribute evaluation algorithm, which will be given later. But this extension will be an interesting theme for future improvement.

3. Incremental attribute evaluation

In this section, we show a method of incremental attribute evaluation based on a class of one-pass attribute grammars.

As a running example, we use the following attribute grammar. Its syntactic part is the same as in grammar G1. The attribute *lev* represents the number of enclosing parentheses in an expression.

AG1: (0) $E' \rightarrow E$
{ E.lev = 0 }
(1) $E \rightarrow E + T$
{ E_2.lev = E_1.lev ; T.lev = E_1.lev }
(2) $E \rightarrow T$
{ T.lev = E.lev }
(3) $T \rightarrow T * F$
{ T_2.lev = T_1.lev ; F.lev = T_1.lev }
(4) $T \rightarrow F$
{ F.lev = T.lev }
(5) $F \rightarrow (E)$
{ E.lev = F.lev + 1 }
(6) $F \rightarrow i$
{/* here F.lev is the no. of parentheses enclosing i */}

Subscripts like E_1, E_2 etc. are used to discriminate occurrences of grammar symbols in productions.

Incremental attribute evaluation presented here is based on a class of one-pass attribute grammars called ECLR-attributed grammar [Sassa 87]. It is a class of attribute grammar where attribute evaluation can be made in one-pass during LR parsing and in a space-efficient way. We first give a brief outline of LR- and ECLR-attributed grammars.

Hereafter, we assume that k of LR(k) is 1. (So, $|v_i|$ is in fact ε, although we retained v_i's in figures.)

3.1 LR-attributed grammar

Suppose that the input for grammar AG1 is

$$i_1 \ {*}_2 \ i_3 \ {+}_4 \ i_5 \ {*}_6 \ i_7 \ {+}_8 \ i_9$$

as in Fig. 5 and the analyzer is now at the beginning, i.e. the lookahead is i_1. Although we do not have the parse tree of Fig. 5 yet since we are at the very beginning, the LR theory tells that the parser is at LR state I_0 of Fig. 3.

Furthermore, since we know the current LR state, it is possible to get the values of inherited attributes even if we do not know exactly the parse tree. For example, we know that in LR state I_0, LR items (i) and (ii) derive (ii) and (iii), (iii) and (iv) derive (iv) and (v), (v) derives (vi) and (vii). Tracing these derivations in reverse order, we are able to see that F.lev = 0, because

$$
\begin{aligned}
F^{5,2}.lev &= T^{5,1}.lev \\
&(= T^{4,2}.lev = T^{4,1}.lev \ldots) \\
&= T^{3,2}.lev = E^{3,1}.lev \\
&(= E^{2,2}.lev = E^{2,1}.lev \ldots) \\
&= E^{1,2}.lev = 0
\end{aligned}
$$

(Attributes in parentheses may or may not occur.) Similarly, we can see that T.lev = 0, because T.lev is either $T^{3,2}$.lev or $T^{4,2}$.lev (note that we can not distinguish between them without actual parse tree) and

$$
\begin{aligned}
T^{3,2}.lev &= E^{3,1}.lev \\
&(=E^{2,2}.lev = E^{2,1}.lev \ldots) \\
&=E^{1,2}.lev =0
\end{aligned}
$$

and

$$
\begin{aligned}
T^{4,2}.lev &= T^{4,1}.lev \ (= T^{4,2}.lev = T^{4,1}.lev \ldots) \\
&= T^{3,2}.lev = \text{(as above)} \\
&= 0
\end{aligned}
$$

Thus, we are able to know that F.lev of F somewhere above i_1 of Fig. 5 (F_1, in this case) is 0 and T.lev of T somewhere above i_1 (T_1 in this case) is also 0.

Note that we have been able to get the values of inherited attributes even if we do not know the exact parse tree. This is the basic idea of LR-attributed grammar (henceforth LR-AG). That is, an LR-AG is known to be a class of attribute grammars where the values of inherited attributes can be computed "uniquely", or without any inconsistency, during LR parsing [Jones 80] [Sassa 85b].

In LR-AG, evaluation of inherited attributes is made at the point when the parser enters a new LR state, that is, at state transition time. This means that we can make "semantic action" (in traditional terminology) not only at reduction time, but also in the midst of the right hand side of a production.

A more complete description of LR-AG can be found in [Sassa 85b].

3.2 ECLR-attributed grammar

In the previous section, readers would have noticed that most values of attribute *lev* of AG1 are the same. For example in LR state I_0, the values of E.lev, T.lev and F.lev are all the same. We can utilize this characteristic to save storage space and evaluation time for inherited attributes as follows.

We collect the set of inherited attributes which have the same value in each LR state into an equivalence class. For example in AG1, we can make an equivalence class

$$EC_1 = \{ E.lev, T.lev, F.lev \}$$

In storing attribute values, we allocate a single location not for each inherited attribute but for each equivalence class. This is the basic idea of ECLR-attributed grammar (hereafter ECLR-AG). Introduction of equivalence classes contributes to reduction of storage space for inherited attributes. A space reduction of 1/17 - 1/9 is reported in [Sassa 87]. Also, a time reduction of about 8 percent is reported there.

To define ECLR-AG more formally, we introduce some concepts.
First, the L-attributed property is defined as usual.

Def. Attribute grammar AG is called **L-attributed**, iff for any production $X_0 \to X_1 \ldots X_{np}$ the following condition holds.
Each inherited attribute of X_k ($1 \le k \le np$) depends only on inherited attributes of X_0 and synthesized attributes of $X_1 \ldots X_{k-1}$.

Next, let EC = { EC_1, EC_2, ... , EC_n } be a disjoint partition of the set of all inherited attributes of a given grammar. Each EC_i is called an **equivalence class**. An equivalence class is supposed to be a set of inherited attributes whose values are mutually the same in each LR state. For example, we may let EC = { EC_1 }, EC_1 = { E.lev, T.lev, F.lev } for grammar AG1.

Then, let **IN** be the set of inherited attributes of nonterminals after the "." (dot or the LR marker) of LR items in a given LR state. It represents the set of inherited attributes to be evaluated at that LR state. That is, if I_i is an LR state,

$$IN(I_i) = \{ A.a \mid A.a \text{ is an inherited attribute of } A, A \text{ is a nonterminal such that } [B \to \alpha \, . \, A \, \beta] \text{ is an LR item of } I_i \}.$$

For example, $IN(I_0)$ of the above LR state I_0 is { E.lev, T.lev, F.lev }.

Lastly, in order to describe that attribute values can be evaluated "uniquely", we introduce a function called semantic expression. Since this concept is important in defining ECLR-AGs, we explain it in detail.
Recall that we got

F.lev =0 and T.lev =0

in the example before. In general, we can see that the value of an inherited attribute $A.a$ in IN(I_i) for an LR state I_i can be computed as a function of the values of attributes in the kernel of I_i. This function is called the semantic expression [Jones 80, Sassa 85b, Sassa 87]. That is, the **semantic expression** $E_{Ii}(A.a)$ of an inherited attribute $A.a$ in IN(I_i) of LR state I_i is a set of possible expressions or symbolical forms for evaluating $A.a$ in terms of attributes of LR item(s)in the kernel of I_i. For example,

$$E_{I0}(\text{F.lev}) = \{\text{expr. for evaluating } F^{5,2}.\text{lev}\} = \{0\}$$

$$E_{I0}(\text{T.lev}) = \{\text{expr. for evaluating } T^{3,2}.\text{lev}\} \cup \{\text{expr. for evaluating } T^{4,2}.\text{lev}\} = \{0\} \cup \{0\} = \{0\}$$

Similarly, we can see that

$$E_{I0}(\text{E.lev}) = \{0\}$$

Since $E_{I0}(A.a) = \{0\}$ for all $A.a \in \text{IN}(I_0) \cap EC_1$, we denote this by

$$E_{I0}(EC_1) = \{0\}$$

The fact that an inherited attribute value is evaluated uniquely can be expressed by that the semantic expression contains only one expression.

Let us look at one additional example.

In LR state I_4 of Fig. 3(a), we see that the value of $E^{1,3}$.lev is

$$E^{1,3}.\text{lev} = F^{1,1}.\text{lev} + 1$$

according to the associated semantic rule. Also, the value of $E^{2,2}$.lev is

$$E^{2,2}.\text{lev} = F^{1,1}.\text{lev} + 1$$

since

$$E^{2,2}.\text{lev} = E^{2,1}.\text{lev} = (E^{2,2}.\text{lev} = E^{2,1}.\text{lev} = \ldots) = E^{1,3}.\text{lev} = F^{1,1}.\text{lev} +1$$

from the derivation of LR items.

How can the value of $F^{1,1}$.lev be obtained? Considering that F.lev $\in EC_1$, and the LR marker is at the second position of the right-hand side of the LR item

$$F^{1,1} \to (\,.\, E^{1,3}\,)$$

which is in the kernel of LR state I_4, we see that $F^{1,1}$.lev is obtained from the value of equivalence class EC_1 at the second position from the top of the current viable prefix. Let us denote this fact by

$$(EC_1, -1)$$

where "-1" means that it is at "top-1" in the current viable prefix.

Thus, the semantic expression of E.lev in LR state I_4 becomes

$$E_{I4}(\text{E.lev})$$

$$=\{\text{expr. for evaluating } E^{1,3}.\text{lev}\} \cup \{\text{expr. for evaluating } E^{2,2}.\text{lev}\}$$

$=\{$expr. for $F^{1,1}.lev + 1 \} \cup \{$expr. for $F^{1,1}.lev + 1\}$
$=\{ (EC_1, -1) + 1 \} \cup \{ (EC_1, -1) +1 \}$
$=\{ (EC_1, -1) +1 \}$

Similarly, we can see that

$$E_{I4}(A.a) = \{ (EC_1, -1) +1 \}$$

for all $A.a \in IN(I_4) \cap EC_1 = \{$ E.lev, T.lev, F.lev $\}$. Putting them all together, we get

$$E_{I4}(EC_1) = \{ (EC_1, -1) + 1 \}$$

We show all semantic expressions for LR states of Fig. 3(a) in Fig. 3(b).

Now, the definition of ECLR-AG is as follows.

Def. A grammar G is **ECLR-attributed** with respect to a partition $EC = \{ EC_1, EC_2, \ldots , EC_n \}$, iff
(1) G is L-attributed, and
(2) for each EC_i, and for each LR state I_j of G, semantic expressions $E_{I_j}(A.a)$'s are the same and unique (i.e. contain only one expression) for all inherited attributes $A.a \in EC_i \cap IN(I_j)$.

In this report, we assume that the partition is given by the user. (Although there is a study about automatic partition of inherited attributes into equivalence classes [Yamashita 87], it will be relatively independent of the current subject.)

<u>Example</u>: Grammar AG1 is ECLR-AG with respect to $EC = \{ EC_1 \}$, $EC_1 = \{$ E.lev, T.lev, F.lev $\}$, since
(1) AG1 is L-attributed, and
(2) for LR state I_0, $E_{I0}(A.a)$'s are $\{ 0 \}$ and are the same and unique for all inherited attributes $A.a \in EC_1 \cap IN(I_0) = \{$ E.lev, T.lev, F.lev $\}$. Similar reasoning holds for other LR states.

A more complete description of ECLR-AG can be found in [Sassa 87].

Note: Some readers may be interested in the descriptive power of ECLR-AG (or LR-AG since they are similar) compared with other classes of attribute grammars. We found from our experience of making translators of several languages using Rie that writing descriptions in the ECLR-AG form was as easy as writing these using L-attributed grammars. We can utilize the information at the left of the point of analysis through inherited attributes, as in L-attributed grammars. Since we want to evaluate attributes during left-to-right parsing, the L-attributed property, which is condition(1) in the definition of ECLR-AG (or LR-AG), is a property that must be inherently satisfied. The additional condition (2) in the definition turns out to be weak. Condition (2) allows copy rules for inherited attributes or non-copy rules for them at the second to last grammar symbols in the right hand side of a production. In many programming languages, inherited attributes such as environment are either simply copied or modified only in block entry to form a nested scope. And a natural description of both cases can easily satisfy this condition(2).

3.3 The normal evaluator

In this section, we show the normal evaluator based on ECLR-AG, which both parses the input and evaluates attributes, making an APT.

Attribute storage in APT

In the last section, we saw that in ECLR-AGs we can allocate storage for each equivalence class at an LR state. This means that in the parse tree, we need not store values of inherited attributes in every node. Rather, we can store them only in some nodes corresponding to some LR states, using a single location for all inherited attributes in the same equivalence class.

Let us call LR states in which $\mathrm{IN}(I_i)$ is not empty *evaluation states*. For example in Fig. 3(a), I_0, I_4, I_6 and I_7 are evaluation states. In the APT, we allocate storage for equivalence classes to nodes which have those evaluation states as their "next" LR states. Here, "next" state of a node means the LR state to which the parser makes transition after reading the grammar symbol corresponding to that node in that parsing configuration. Note that a "next" LR state of a node depends on the context. Let us also add a special node ¢ into the APT, which has the initial LR state I_0 as the "next" LR state.

As an example, in the APT of Fig. 5(b), we allocate storage for equivalence classes to nodes ¢ (which has the evaluation state I_0 as the "next" state), $*_2$(-"- I_7), $+_4$(-"- I_6),$*_6$(-"- I_7) and $+_8$(-"- I_6). They have values for EC_1 = { E.lev. T.lev, F.lev } which are all 0 in this particular example. Here only a small part of nodes contain values of inherited attributes.

Let us mention about possible data structures. For inherited attributes corresponding to an evaluation state I_i, we can use a field of record like

inh_attr_I_i : **record** ec_i1: ...; ec_i2: ...; ec_i3: ...; ... **end**

Fields ec_i1 etc. correspond to equivalence classes. This record may vary from evaluation state to evaluation state, and only those equivalence classes which actually appear in IN(I_i) will have thier fields. For the running example, the data structure will be

inh_attr_I_0 or inh_attr_I_4 or inh_attr_I_6 or inh_attr_i_7:

record ec_1: integer **end**

As for synthesized attributes, storage is allocated as usual in nodes of the APT, of which the corresponding grammar symbol has synthesized attributes. Storage for synthesized attributes in node X_i might be a field like

syn_attr_X_i : **record** X_i_syn_1: ...; X_i_syn_2: ...; X_i_syn_3: ...; ... **end**

Note : An equivalence class can also contain some independent set of inherited attributes as noticed in [Sassa 87]. In that case, the type of ec_i1 etc. may be a union of several different types.

The normal evaluator

Let us now present the normal evaluator which, in addition to parsing and making the parse tree, evaluates attributes and stores their values into nodes of the parse tree, making the APT. (Note : The evaluator presented here is a little different in appearance from the one presented in [Sassa 87], although the principle is the same (see discussion).)

The configuration of the parse stack in this normal evaluator is similar to the one in the incremental parser given before. Only the bottom element is a little different. The form of the parse stack is in general

$(\text{nil}, p_{¢})\ I_0\ (X_1, p_{X1})\ I_1\ (X_2, p_{X2}) \cdots (X_m, p_{Xm})\ I_m$

where I_i is an LR state, X_i is a grammar symbol and p_{Xi} is a pointer to the node in APT corresponding to X_i. In particular, $p_{¢}$ is a pointer to node ¢.

Note: As in section 2, X_i's need not be stored.

Now, the algorithm for the normal evaluator is as follows.

Algorithm Normal evaluator
Input: The input $w = x_0 y_1 x_1 y_2 x_2 \cdots y_m x_m$.
Output: APT of w .
Method:
configuration := ((nil, $p_{¢}$) I_0 , $a_1 \dots a_n$ \$) ;
loop
  let configuration **be**
    ((nil, $p_{¢}$) I_0 (X_1 , p_{X1}) ... I_{m-1} (X_m , p_{Xm}) I_m , $a_j \dots a_n$ \$) ;
  action := ACTION [I_m , a_j] {ACTION in the parse table} ;
  if action = "accept" or action = "error" **then** exit ;
  if IN(I_m) ≠ Ø then compute values of equivalence classes of inherited

attributes in $IN(I_m)$ {note 2,3} and put them in node pointed by p_{Xm};

case action **of**

"shift I":

make a new (leaf) node corresponding to a_j;

put values of synthesized attributes of a_j {from lexical analysis} into that node ;

p_{aj} := pointer to that node ;

configuration := (... I_m (a_j, p_{aj}) I, a_{j+1} ... a_n \$) ;

"reduce by $A \rightarrow \alpha$":

make a new (internal) node corresponding to A ;

compute values of synthesized attributes of A {note 3}

and put them into that node ; {note 4}

p_A := pointer to that node ;

k := $|\alpha|$;

make the node pointed by p_A be the father of nodes pointed by p_{Xm-k+1}, p_{Xm-k+2}, ... , p_{Xm};

pop configuration down to (... I_{m-k} , a_j ... a_n \$) ;

I := GOTO [I_{m-k} , A] {GOTO in the parse table} ;

configuration := (... I_{m-k} (A, p_A) I , a_j ... a_n \$) ;

end case

end loop

Notes:

1. In this attribute evaluator, evaluation takes place at following moments:

(i) Synthesized attributes of a nonterminal A are evaluated when the parser reduces by a production "$A \rightarrow \alpha$".

(ii) Inherited attributes of a nonterminal B are evaluated when the parser goes to an LR state which contains an LR item [$A \rightarrow \beta$. $B \gamma$] with some A, β and γ. In other words, inherited attributes are evaluated at the time when the parser makes transition to an evaluation state.

2. If the semantic expression for an equivalence class is of the form

$$E_{Im}(EC_i) = \{...(EC_j,-k)...\}$$

the value of $(EC_j,-k)$ can be obtained from storage (of evaluation state) of the node pointed by p_{Xm-k}.

3. Values of synthesized attributes necessary for evaluation can be found from storage of nodes pointed by p_{Xm}, p_{Xm-1}, p_{Xm-2} ,... etc.

4. The attribute matching condition of the incremental evaluator (which will be presented later) should be checked at this point.

Example The normal evaluator for input i * i + i * i + i proceeds as shown in Fig. 8. This makes the APT of Fig. 5(b).

Discussion

In this section, we presented the algorithm for the normal evaluator in a way that (i) attribute values are stored only in nodes of the APT.

It is possible to implement it in a different way by (ii) using attribute stacks which are synchronous with the parse stack, evaluate attributes on those stacks as in [Sassa 87], and at reduction time, copy the elements of attribute stacks popped at that moment into nodes of the APT.

The difference between (i) and (ii) is that in (i) if we want to reduce memory requirements and store attribute values in a packed form, the location of the field for inherited attributes of an equivalence class may vary from node to node and there may be some time overhead to retrieve those attribute values, while in (ii) there is no such problem but there may be possibly more time overhead by copying attribute values from attribute stacks to nodes.

3.4 The incremental evaluator

In this section, we present the incremental evaluator based on the ECLR-AG. We note that the power of the incremental evaluator is naturally the same as the normal evaluator.

The general idea of incremental attribute evaluation is similar to the incremental parser. One difference is that modification of y'_1 may also affect attribute values in some part "above" u_i in addition to the part "above" v_{i-1} and t_i (Fig. 1(b)). That is, there may be some part above u_i where the attribute values become invalid, although the parse tree is valid there. The general scheme is as shown in Fig. 9. The shaded part remains valid concerning the parse tree and attribute values.

Here, t_i and v_i are the same as before, but u_i is now divided into two parts r_i and s_i. We define r_i so that in the part above r_i, the parse tree is the same as the original one, but the attribute values are not the same. In the part above s_i, both the parse tree and the attribute values are the same ($r_i = \varepsilon$ for $i = 0$).

Thus in general, $w' = x_0\, y'_1\, x_1\, y'_2\, x_2 \dots y'_m\, x_m$, $x_i = t_i\, r_i\, s_i\, v_i$.

Now, we are ready to present the incremental evaluator.

The idea is to combine the incremental parser of section 2 and the normal evaluator of section 3.3 with consideration of the validity of attribute values. Two points, initialization and termination should be made clear.

3.4.1 Initialization of the incremental evaluator

First, we will show some properties concerning Fig. 9.

Prop. 3.1 Values of attributes (also values of inherited attributes in evaluation states associated to nodes in this part) in the shaded part "above" s_i (if $s_i \neq \varepsilon$) are still valid after modification.

(informal proof)

The part just to the right of the boundary between r_i and s_i is valid by definition.

For synthesized attributes, we pay attention to the fact that the boundary between s_i and v_i may be zig-zag. However, nodes in the shaded part have all their sons in the shaded part. Since values of synthesized attributes of a node depend only on the sons of that node, it is clear that attribute values of nodes in the shaded part are still valid after modification. For example in Fig. 6, the shaded part above s_0 consists of i_1, F_1, T_1 and $*_2$. Synthesized attributes of T_1 are valid since its subtree F_1 and i_1 is in the shaded part after modification.

For inherited attributes, this property needs some explanation. In general, the shaded part above s_i may not be connected to the root node (cf. Fig. 1(c)). For example in Fig. 6, T_1 is not connected to the root node E' by arcs of the original parse tree.

However, recall that in ECLR-AGs, values of inherited attributes of a node are computed according to the current LR state, and using the attributes of nodes pointed to by the parse stack elemnets (not using the whole parse tree). Let us think of the time of evaluation of inherited attributes of a node n in the shaded part above s_i. At this moment all nodes pointed to from the parse stack reside in the shaded part. So, all nodes pointed to from the parse stack and the LR state at that node n are the same as when we evaluated in the original parse tree. Thus, values of inherited attributes of node n are the same as before. Using induction on the length of the parse stack, we can conclude that the values of inherited attributes of evaluation states associated with nodes of the shaded part are still valid after modification. In this statement, we shall include "node ¢ and its evaluation state I_0" in the shaded part above s_0.

For example in Fig. 6, inherited attributes (which is in ¢) for nodes T_1 and F_1 above s_0 were computed when the parse stack contained only the initial LR state I_0 and the parser was in LR state I_0. The evaluation is independent of nodes T_{13}, T_{14} and E_2. Similarly, inherited attributes (which is in $*_2$) for the node F_{13} are still valid. (end of informal proof)

We also note the following property.

Prop. 3.2 When $s_i \neq \varepsilon$, if the LR state after reading the first symbol of v_i (let it be a) is an evaluation state, values of inherited attributes at that LR state are still valid after modification.

The values of inherited attributes at the "next" state of a are determined by the parse stack and the current LR state. Similarly to Prop 3.1, we can say that neither has changed after modification of input.

Now, initialization of the incremental evaluator for y'_i is quite similar to that of the incremental parser.

procedure Initialize incremental evaluator (for y'_i):

(1) Put (nil, $p_{¢}$), where $p_{¢}$ is a pointer to node ¢, and then the initial LR state I_0 on the bottom of the stack.

(2) Get the prefix chain $a, X_{n-1}, X_{n-2}, \ldots, X_1$ starting from the last terminal symbol a of y_{i-1}.

(3) If the prefix chain = ε, then skip this step.
Otherwise, put into the parse stack each grammar symbol in the above prefix chain in reverse order like $X_1, X_2, \ldots, X_{n-1}, a$ with pointers to the corresponding nodes in the original parse tree $p_{X1}, p_{X2}, \ldots, p_a$, and with recovering the corresponding LR states $I_1, I_2, \ldots, I_{n-1}, I_n$ by performing LR parsing. Thus in general, the parse stack is like

$$(\text{nil}, p_{¢})\ I_0\ (X_1, p_{X1})\ I_1\ (X_2, p_{X2})\ I_2 \ldots (X_{n-1}, p_{Xn-1})\ I_{n-1}\ (a, p_a)\ I_n$$

where p_a is a pointer to the node representing a.

(4) Let the remaining input be

$$b \ldots \$$$

where b is the input symbol next to a.

<u>Example</u> In the running example (Fig. 6), if the incremental evaluator starts at point after $*_2$, the parse stack will be initialized as

$$(\text{nil}, p_{¢})\ I_0\ (T_1, p_{T1})\ I_2\ (*_2, p_{*2})\ I_7$$

where p_{T1} and p_{*2} are pointers to nodes for T_1 and $*_2$, respectively. Notice that we are able to access $IN(I_0)$ attached to ¢ and $IN(I_7)$ attached to $*_2$ tracing pointers from the parse stack.

3.4.2 Termination of the incremental evaluator

Termination of the incremental evaluator for the part y'_i requires checking of attribute values in addition to the matching condition for parsing presented in section 2.3.

Assume that the matching condition holds at reduction "$A \to \alpha$". Let the corresponding node of the new parse subtree be n'_A and that of the original parse tree be n_A (Fig. 2(b)).

Recall that in attribute grammars the only way of passing attribute values from the subtree of n'_A outward is through synthesized attributes of n'_A. Therefore, if values of synthesized attributes of n'_A are the same as those of n_A we can really teminate incremental evaluation for the part y'_i.

If synthesized attributes values of n'_A and n_A are not the same, we should continue incremental evaluation. Several ways might be possible how to continue and when to stop re-evaluation. For the sake of simplicity of the algorithm however, here we only show the simplest method, and leave possible improvements to further discussion.

So, here we continue re-evaluation of inherited and synthesized attributes until the *attribute matching condition* holds.

Attribute matching condition: (Fig. 10)

Let n_A be the node in the original parse tree where the matching condition holds. Assume that a reduction "$C \to \beta$" occurs. Let the corresponding node of the original parse tree be n_C. The condition holds if:

(i) Node n_C is an ancestor of n_A , or n_A itself.

(ii) Newly evaluated values of synthesized attributes of n_C are the same as the old values of synthesized attributes of n_C .

The condition means in general that attributes are to be re-evaluated for nodes in the shaded part "above" r_i of Fig. 10 after the matching condition for incremental parsing is satisfied. (The cases in which r_i extends to v_i or y'_{i+1} etc. are treated properly in section 3.4.3.) Whether or not we rewrite attribute values on the original APT in re-evaluating attributes, is discussed in the next section.

3.4.3 Incremental evaluator

We can now present the incremental evaluator as a whole, which is stated in the folllowing algorithm.

Algorithm Incremental evaluator
Input: The APT of $w = x_0 y_1 x_1 y_2 x_2 \dots y_m x_m$ and still unanalyzed input $w' = x_0 y'_1 x_1 y'_2 x_2 \dots y'_m x_m$.
Output: The APT of w' if w' belongs to L(G), otherwise an error indication.
Method: It consists of the following steps:
(1) Set $i = 1$.
(2) Skip analysis of s_{i-1}. By using the procedure "Initialize incremental evaluator" presented before, set the parse stack to have the same contents as when it has just shifted the last terminal symbol of u_{i-1}.
(3) In the following steps (4) through (8), if "accept" or "error" turns up, go to step (10).
(4) Using the normal evaluator, make parsing and attribute evaluation for the rest of v_{i-1} and y'_i while making a new APT subtree.
(5) After the lookahead is within x_i, continue parsing, attribute evaluation and making the new APT subtree, but test the matching condition every time a reduction occurs.
(6) If in (5), (7) or (8) the matching condition or the attribute matching condition does not hold yet, but the lookahead comes to be within v_i ($i<m$), increment i by one and go to step (4).
(7) When the matching condition holds after reading t_i and at node n_A of the original APT, then replace the subtree of n_A by the new APT subtree for ... $v_{i-1} y'_i t_i$.
(8) Continue attribute re-evaluation (see note 1, 2), but test the attribute matching condition every time values of synthesized attributes of a node have been re-evaluated (including the moment in step (7)).
(9) When the attribute matching condition holds at node n_C, then increment i by one. If $i \le m$, then go to step (2).
(10) Stop.

Note:
1. In actual implementation of step (8), we may either
(i) continue incremental LR parsing, attribute evaluation and making of the APT subtree by the normal evaluator, or
(ii) stop incremental parsing and making of the APT, and only re-evaluate attributes in the old APT.

In case (i), since we continue making the new APT subtree, it is safe in cases of semantic errors. On the other hand we should take attention to adjust the boundary conditions before replacing the subtree of n_C by the new APT subtree, for example, do not forget to copy values of inheritied attributes in the evaluation state connected to n_C into the root of the new APT subtree.

In case (ii), careful treatment is necessary if the lookahead comes to be within v_i while re-evaluating attributes in the old APT. Precaution should also be taken in the case of semantic errors, since the method overwrites attribute values. In the actual implementation it will be easier to modify the existing evaluator based on ECLR-AGs and use it, rather than to make a new tree walk evaluator.

2. In step (8), we assume that synthesized attributes of a node are evaluated after all attributes in its subtree are evaluated. If re-evaluation is made based on ECLR-AG, this corresponds to reduction time.

Example: Moves of the incremental evaluator and the resulting APT for the modified input i * (i + i) * i + i are shown in Fig. 11 and Fig. 6(b), respectively.

3.5 Discussion

In the method of attribute evaluation presented here, space for inherited attributes seems to be fairly small. For example in Fig. 5(b), only 5 nodes out of 25 nodes have a storage for inherited attributes. The space reduction comes from two factors. First, the use of LR-attributed grammars makes storage for inherited attributes be allocated only into "evaluation states", not into every node. This realizes some storage optimization, particularly in the case of left-recursive productions. Secondly, the use of equivalence classes in ECLR-AGs makes it possible for inherited attributes in the same equivalence class to share storage, which is more significant.

Several optimization of the method shown here will be possible:

For unit productions, we can omit intermediate nodes of APT if (i) the production is a unit production, (ii) attribute evaluation rules for synthesized attributes of that production are copy rules, and (iii) there is no evaluation state associated with that production.

Also, optimization of incremental evaluation by skipping analysis of some subtrees of x_i in the APT as for incremental parsing will be an interesting problem.

The attribute matching condition for the incremental evaluator might be too restrictive. Reducing the part of re-evaluation at step (8) of the incremental evaluator will be profitable.

In actual attribute grammars, efficient treatment of big values, like the symbol table, should be further investigated [Hoover 86].

In application to language-based editors, the relation between lexical level changes in units of characters and grammar level changes in units of tokens should be considered carefully. For example, a token may be divided into two by a character mode editing.

4. Conclusion

A method of incremental attribute evaluation and parsing is described. It is based on a class of LR-attributed grammars called ECLR-attributed grammars. The method unifies incremental attribute evaluation and incremental parsing in a single algorithm. Multiple modifications in the original input are also allowed.

From the one-pass nature and the use of equivalence classes in ECLR-attributed grammars, reduction of evaluation time and memory size can be expected. In particular, use of equivalence classes contributes quite much to space efficiency of the attributed parse tree. Inherited attributes are stored only in a small part of the nodes of the attributed parse tree, and the storage ruquirements would be 1/3 - 1/10 compared to naive methods.

Acknowledgment

The author would like to thank Eduard Klein of GMD in Karlsruhe for introducing incremental parsing to him, and to Kai Koskimies, Jorma Tarhio, Niklas Holsti of the University of Helsinki, Merik Meriste of Tartu University, Rieks op den Akker of Twente University, and Ikuo Nakata of the University of Tsukuba for helpful discussions.

Appendix 1

About data structures of the APT or the parse tree

An example data structure of the APT is given here. Also, we want to show that it is possible to make the APT space efficient, because it is often said that such internal data structure takes about one order of magnitude more storage compared to that for input.

A possible data structure is given in Fig. A1. An *internal* node in the APT is made as follows:

type internalnode = **record** *symbol*: ... ; *left_son*: ... ; *next*: ... ; *mark*: ... ; $syn\_attr\_X_i$: ... **end**

Namely, an internal node is made of fields *symbol* for storing the corresponding grammar symbol, *left_son* which points to its leftmost son, *next* which points to its right brother (if one exists) or its father (otherwise), together with some *mark* bit to indicate which of these two kinds of pointer *next* is. $syn\_attr\_X_i$ is a field for storing values of synthesized attributes (section 3.3).

LR states need not be stored in the APT (or the parse tree), which may also contribute to space efficiency.

A *leaf* node of the APT is a node for a terminal symbol (token) or ε. Such a node may be of the type similar to internal node, but the following should be taken into account.

(i) For the text editing, it should be possible to access the APT from the input string.

(ii) If there are ε-productions, a symbol in the input may correspond to several leaf nodes including nodes for ε's. In that case, we make a list of ε-nodes and a normal leaf node, and let an input symbol or a character point to that list. Note that ε's belong to the APT (or the parse tree) but not to the input (Fig. A1).

Using the above data structure, we can easily implement *prefix* and *ancest* as functions. In this case, since we do not allocate fields for *prefix* and *ancest*, calling such functions to check the matching condition etc. will take more than constant time. On the other hand, they could be implemented as fields, but then it would increase space requirements.

Notes to the Algorithm "Incremental parser" concerning the data structure:

1. In steps (4) and (5), if a reduction "$B \to \beta$" occurs and if part of nodes for β belong to the original parse tree, precaution is necessary in dealing with the data structure. We should not destroy the part of the original parse tree until the incremental parsing succeeds without arising errors. Also some refinements on the data structure should be necessary.

Pointers from the input string to the leaves of the parse tree should not destroy the original parse tree. Caution is also needed in the relation between the input string and ε-leaf nodes.
2. If we use actual fields for *ancest()* and *prefix()*, at step (7) we should adjust *ancest(c)* and *prefix(l)* for all *l* in nodes of the left border (which may be zig-zag) of the new parse subtree for $\ldots v_{i-1}y'_i t_i$ (and sometimes for nodes of the left border of the original parse tree above *d*, depending on what data structures are used).

Several alternative data structures can be considered.
(i) We can allocate the *ancest* field in each leaf node, and also add a field, say *right*, to each node in the parse tree which points to the leaf node which is the rightmost descendant. At reduction time, we can set the contents of those fields in constant time. To check the matching condition, we can get the node pointed by *ancest* also in constant time. Instead, the space requirements will be larger, and the increase will be proportional to (no. of nodes + no. of input symbols), neglecting ε-nodes.
(ii) We can also allocate the *prefix* field in each leaf node in addition to the *ancest* field, as in [Yeh 88]. This will increase space requirements, and also increase the constant factor (coefficient) of time requirements in the normal and the incremental parsers. But the average time complexity of the incremental parser will be $O(|y'_1 + y'_2 + \ldots + y'_m|)$.
(iii) Compared to the above alternatives, the data structure suggested here is based on the following principles:
- Space requirements for the parse tree and input is small. If we suppose the existence of the parse tree which is naturally necessary in language-based editors dealing with program structures, the data structure proposed here is in fact the one which requires almost minimum space.
- There is no time overhead in the normal parser.
- Some time overhead exists in checking the matching condition in the incremental parser. Its time complexity is roughly $O(h \times |y'_1 + y'_2 + \ldots + y'_m|)$, where *h* is the height of the parse tree.

Appendix 2

Proof of the matching condition

(proof)

If the contents of the "parse stack" and the "remaining input" are the same for the original parsing and the current parsing, future moves of the parser (in part x_i except v_i) will be the same.

The equality of the remaining input (in part x_i except v_i) is clear from condition (i).

The equality of the parse stack can be deduced from conditions (ii) and (iii). The proof is given in (4) in the following, but before that we note properties (1) - (3).

(1) The parse stack contains the viable prefix or a prefix of the right sentential form which is determined uniquely if we specify a node in the parse tree. The prefix chain made of prefix() operation specifies exactly this viable prefix [Yeh 88]. Moreover, LR states in the parse stack are completely determined if we have a viable prefix in the parse stack.

(2) The property to be shown here concerns the content of the parse stack. In the following, we omit the boundary case where only "(nil, nil) I_0" is in the parse stack.

The parse stack was

$$(\text{nil},\text{nil})\ I_0\ (X_1, p_{X1})\ I_1\ (X_2,p_{X2})\ I_2 \ldots (X_{n-1}, p_{Xn-1})\ I_{n-1}\ (a, p_a)\ I_n \qquad \text{(a)}$$

at the time of initialization. $X_1,\ldots,X_{n-1},a$ is a viable prefix from property (1). Pointers $p_{X1}, p_{X2},\ldots, p_{Xn-1}$, all point to nodes "above" u_{i-1} or the part to the left of it in the original parse tree.

As parsing of $v_{i-1}y'_ix_i$ proceeds, the parse stack changes to a form

$$(\text{nil},\text{nil})\ I_0\ (X_1, p_{X1})\ I_1 \ldots (X_k,p_k)\ I_k\ (X'_{k+1},p_{X'k+1})\ I'_{k+1} \ldots (X'_m,p_{X'm})\ I'_m$$

Since new elements put on the parse stack are by "shift"s and "reduce"s, they correspond to nodes in the new parse subtree. ..(b)

Thus,

- $p_{X1}, p_{X2}, \ldots, p_{Xk}$ are the same as in (a) and all point to nodes above u_{i-1} or the part to the left of it in the original parse tree, and
- $p_{X'k+1}, \ldots, p_{X'm}$ all point to nodes in the new parse subtree.

From property (b), the part of the parse stack elements corresponding to the original parse tree is always decreasing (or precisely, not increasing).

(3) We show that n_A and n'_A really covers the part y_i and y'_i, respectively, i.e., it is not like n^*_A or n'^*_A in Fig. 2 (a)(c). (This property is not needed in the proof (4)).

When condition (iii) of the matching condition holds, the parse stack is like

(nil,*nil*) I_0 (X_1, p_{X1}) I_1 . . . . (X, p_X) I_{q-1} (A, p'_A) I_q (c)

From note 1 of the text, p_X points to a node in the original parse tree. From this and property (2) above, (X, p_X) should have existed in (a) at the time of initialization. Thus, p_{X1} ,..., p_X in (c) are the same as in (a) and all point to nodes above u_{i-1} or the part to the left of it in the original parse tree. So n_A and n'_A covers the part y_i and y'_i, respectively.

(4) Now, we prove the equality of the parse stack for the original and the current parsing.

When condition (iii) of the matching condition holds, the parse stack is like

(nil,nil) I_0 (X_1, p_{X1}) I_1 ... (X_{q-2}, p_{Xq-2}) I_{q-2} (X, p_X) I_{q-1} (A, p'_A) I_q (c)

When the original input was parsed, the parse stack at n (when nonterminal for n was reduced) was a viable prefix like

(nil,nil) I_0 (Y_1, p_{Y1}) J_1 ... ... (Y_{r-1}, p_{Yr-1}) J_{r-1} (A, p_A) J_r (d)

p_A corresponds to n in the matching condition.

From the characteristics of prefix(), p_{Yi} = prefix (p_{Yi+1}) (i=1, ..., r-2) and p_{Yr-1} = prefix(p_A). Y_i is the grammar symbol of the node pointed by p_{Yi} (i=1,...,r-1).

Since p_X = prefix(n) by condition (iii), it follows that p_X = prefix(n) = prefix(p_A) = p_{Yr-1}. From $p_X = p_{Yr-1}$ and property (1), $q = r$, $p_{Xi} = p_{Yi}$, thus $X_i = Y_i$ (i=1,...,q-2). Also from property (1), $I_i = J_i$ (i=1,...,q-1), $I_q = J_r$. Since we replace p'_A by p_A after the matching condition holds, all the elements of (c) = (d). (Q.E.D)

Appendix 3

Notes on ε-productions in the matching condition

E1. We can also check the matching condition even if α is ε, since the condition may hold in some cases, e.g., $y'_i = \varepsilon$, "$A \to \varepsilon$" $\in P$ and $A \Rightarrow^* y_i$.

E2. Precisely, c is a leaf node rather than a symbol. If there are several ε-nodes before d in the original parse tree, which have been reduced to some nonterminals, as

$$\begin{array}{ccccc} & X & Y & Z & \\ | & | & | & | & \\ f & \varepsilon_1 & \varepsilon_2 & \varepsilon_3 & d \end{array}$$

we let c in the matching condition be $f, \varepsilon_1, \varepsilon_2, \varepsilon_3$ in turn, and check all of them. This means that the boundary between t_i and u_i has the following possibilities:

$$\begin{array}{ll} t_i = \ldots c & u_i = \varepsilon_1\, \varepsilon_2\, \varepsilon_3\, d\ldots \\ t_i = \ldots c\, \varepsilon_1 & u_i = \varepsilon_2\, \varepsilon_3\, d\ldots \\ t_i = \ldots c\, \varepsilon_1 \varepsilon_2 & u_i = \varepsilon_3\, d\ldots \\ t_i = \ldots c\, \varepsilon_1 \varepsilon_2\, \varepsilon_3 & u_i = d\ldots \end{array}$$

However, we can omit those complicated checks, if we do not care to stop incremental parsing at the earliest point, since the matching condition will soon hold anyway.

E3. When replacing the subtree of n_A by the new parse subtree, some treatments would be necessary for adjusting the ε-leaf nodes in the data structure.

References

[Agrawal 83] Agrawal, R. and Detro, K.D. An Efficient Incremental LR Parser for Grammars With Epsilon Productions, Acta Inf. 19, 369-376 (1983).

[Aho 86] Aho, A.V., Sethi, R. and Ullman, J.D. Compilers - Principles, Techniques, and Tools, Addison-Wesley, 1986.

[Celentano 78] Celentano, A. Incremental LR Parsers, Acta Inf. 10, 307-321 (1978).

[Ghezzi 80] Ghezzi, C. and Mandrioli, D. Augmenting Parsers to Support Incrementality, J. ACM, 27, 3, 564-579 (1980).

[Hoover 86] Hoover, R. and Teitelbaum, T. Efficient Incremental Evaluation of Aggregate Values in Attribute Grammars, Proc. ACM SIGPLAN '86 Symp. on Compiler Construction, 39-50 (1986).

[Ishizuka 85] Ishizuka, H. and Sassa, M. A Compiler Generator Based on Attribute Grammars (in Japanese), Proc. 26th Programming Symposium, IPS Japan, 69-80 (1985).

[Jalili 82] Jalili, F. and Gallier, J.H. Building Friendly Parsers, 9th ACM Symp. on POPL, 196-206 (1982).

[Jones 80] Jones, N.D. and Madsen, M. Attribute-influenced LR Parsing, Lecture Notes in Comp. Sci. 94, 393-407 (1980).

[Knuth 68] Knuth, D.E. Semantics of Context-Free Languages, Math. Syst. Th. 2, 2, 127-145 (1968), correction *ibid.* 5, 1, 95-96 (1971).

[Koskimies 88] Koskimies, K., Nurmi, O., Paakki, J. and Sippu, S. The Design of a Language Processor Generator, Softw. Pr. Exper. 18, 2, 107-135 (1988).

[Notkin 85] Notkin, D. The GANDALF Project, Jour. Syst. Softw. 5, 91-105 (1985).

[Reps 83] Reps, T., Teitelbaum, T. and Demers, A. Incremental Context-Dependent Analysis for Language-Based Editors, ACM Trans. Prog. Lang. Syst. 5, 3, 449-477 (1983).

[Sassa 85a] Sassa, M. Ishizuka, H. and Nakata, I. A Compiler Generator Based on LR-attributed Grammars, Tech. Memo PL-7, Inst. of Inf. Science, Univ. of Tsukuba, 1985.

[Sassa 85b] Sassa, M. Ishizuka, H. and Nakata, I. A Contribution to LR-attributed Grammars, Jour. Inf. Process. 8, 3, 196-206 (1985).

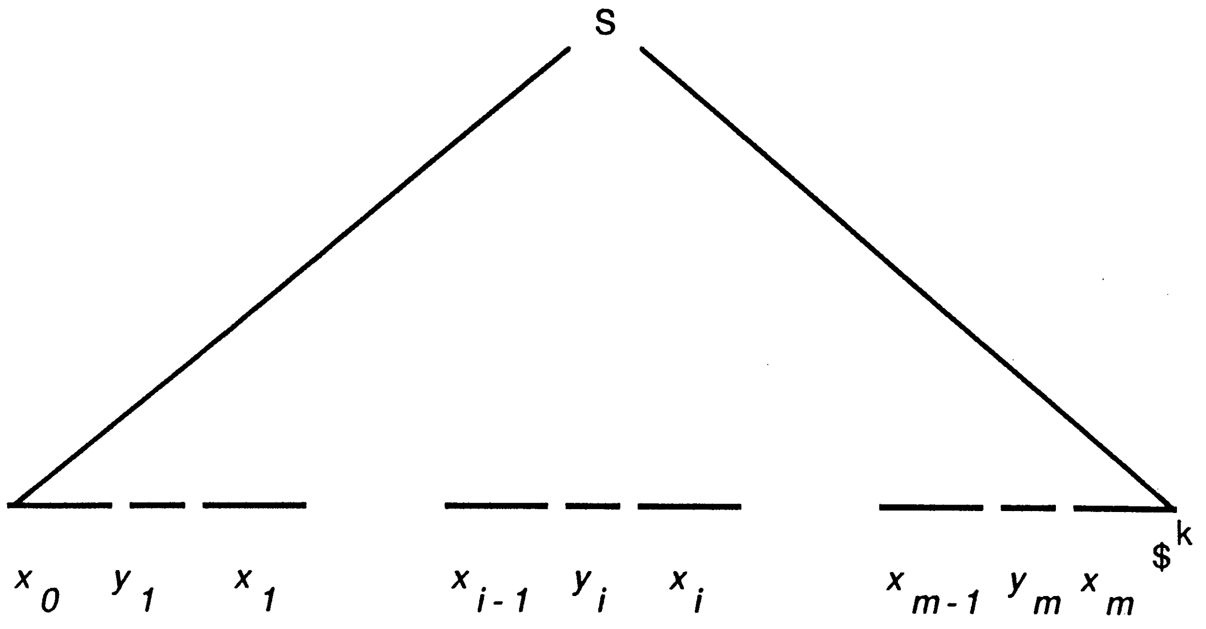
[Sassa 87] Sassa, M. Ishizuka, H. and Nakata, I. ECLR-attributed Grammars: A Practical Class of LR-attributed Grammars, Inf. Process. Lett. 24, 31-41 (1987).

[Yamashita 87] Yamashita, Y., Sassa, M. and Nakata, I. A Friendship Club Problem and its Applications to Attribute Grammars (in Japanese), Computer Software 4, 3, 28-40 (1987).

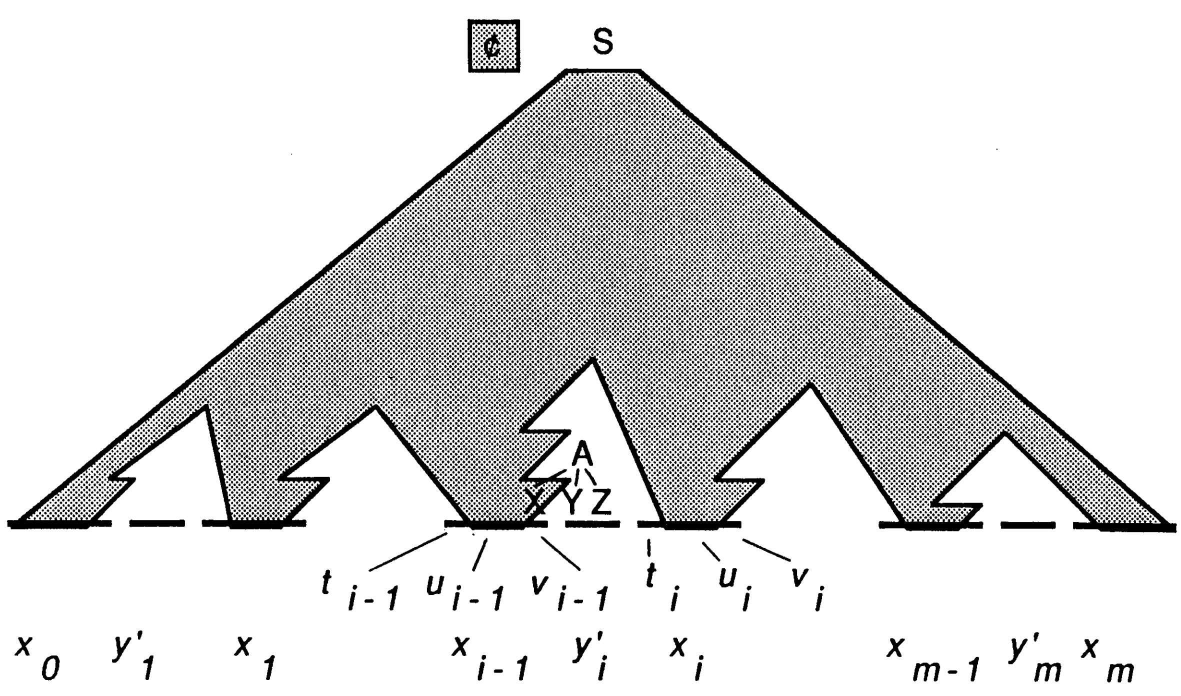
[Yeh 83a] Yeh, D. On Incremental Shift-Reduce Parsing, BIT 23, 36-48 (1983).

[Yeh 83b] Yeh D. On Incremental Evaluation of Ordered Attributed Grammars, BIT 23, 308-320 (1983).

[Yeh 88] Yeh, D. and Kastens, U. On Mechanical Construction of Incremental LR(1) Parsers, draft, Tongji Univ., Shanghai, P.R. China (1988).



(a)



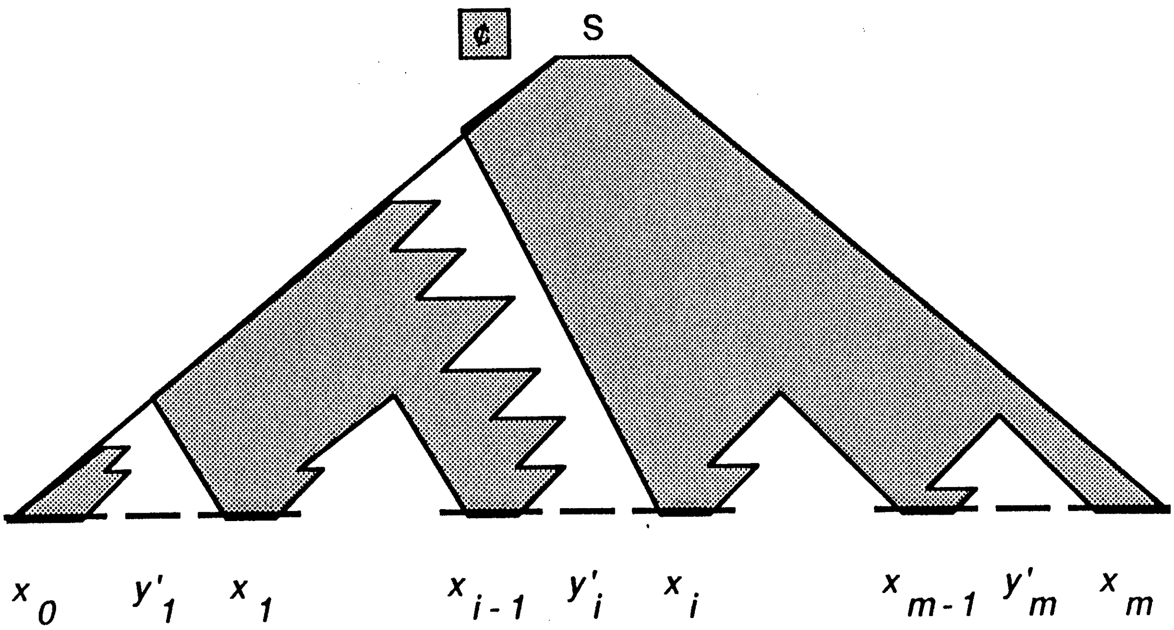
Shaded part is valid. ¢ will be explained later.

(b)

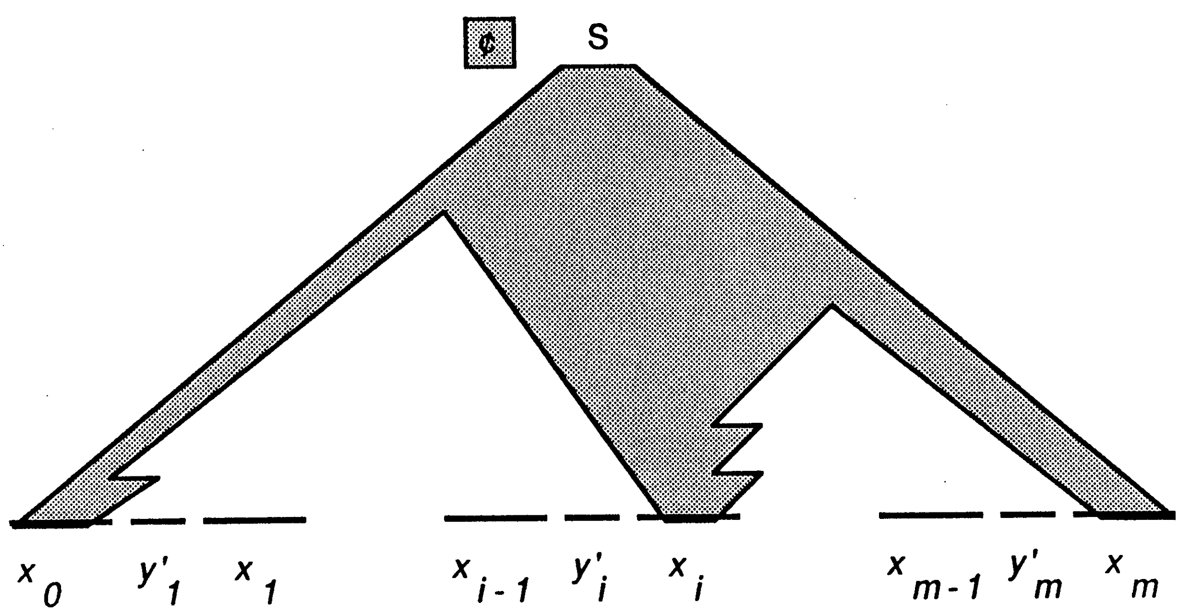
(a) Original parse tree

(b)(c)(d) Modified parse tree

Fig. 1 Original and modified parse tree

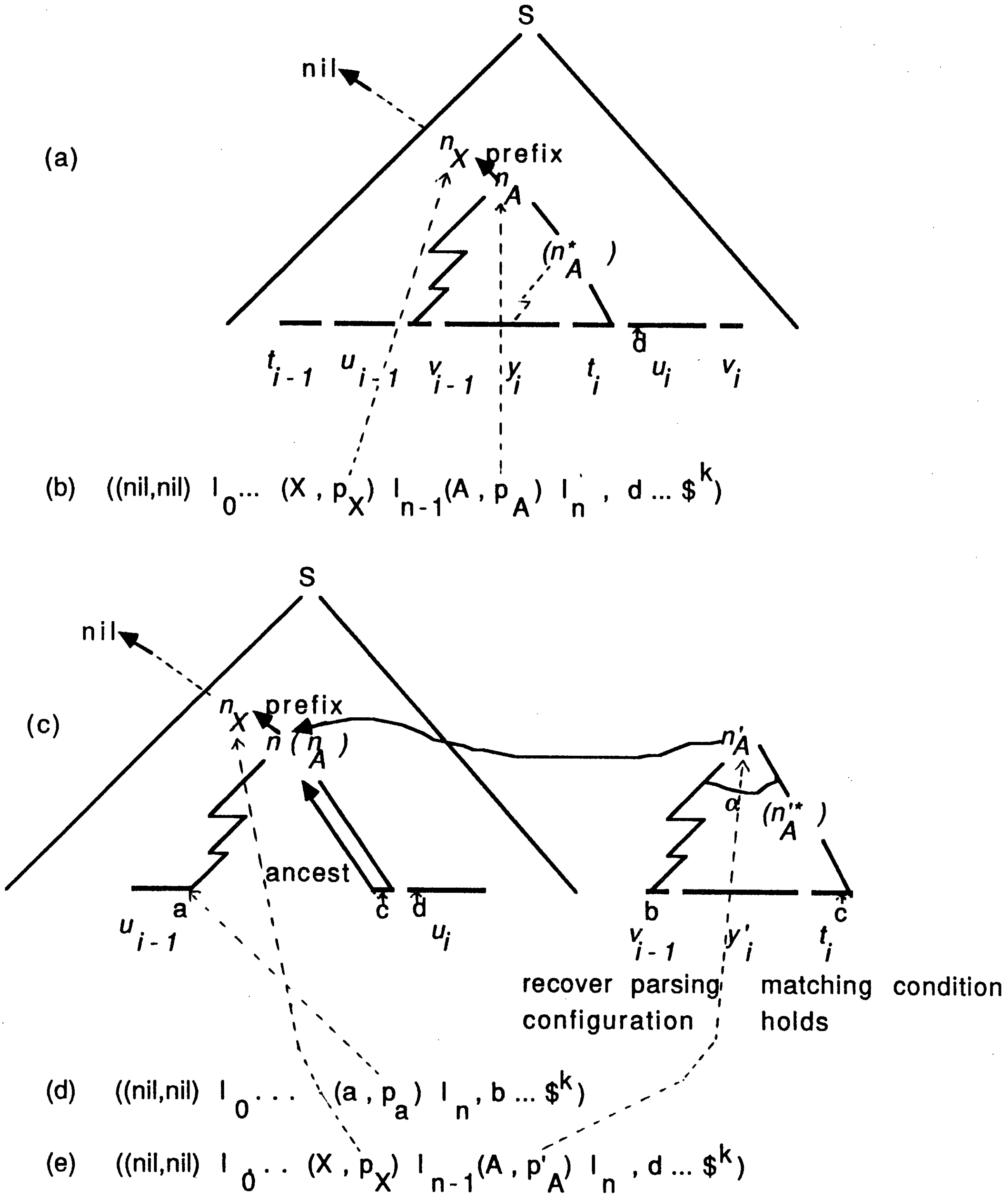


(c)



(d)

Fig. 1 (cont.)



(a) original parse tree

(b) original parsing configuration just before d of (a)

(c) original parse tree (left) and new parse subtree (right)

(d) initialization of parsing configuration

(e) current parsing configuration just before d of (c)

Fig. 2 Matching original and modified parse tree

I_0:	E'	$\rightarrow$	$. E^{1,2}\ \$$	(i)	$E_{I_0}(EC_1) = \{ 0 \}$
	$E^{2,1}$	$\rightarrow$	$. E^{2,2} + T$	(ii)	
	$E^{3,1}$	$\rightarrow$	$. T^{3,2}$	(iii)	
	$T^{4,1}$	$\rightarrow$	$. T^{4,2} * F$	(iv)	
	$T^{5,1}$	$\rightarrow$	$. F^{5,2}$	(v)	
	$F^{6,1}$	$\rightarrow$	$. (E)$	(vi)	
	$F^{7,1}$	$\rightarrow$	$. i^{7,2}$	(vii)	

I_1:	E'	$\rightarrow$	$E .$	
	E	$\rightarrow$	$E . + T$	
I_2:	E	$\rightarrow$	$T .$	
	T	$\rightarrow$	$T . * F$	
I_3:	T	$\rightarrow$	$F .$	
I_4:	$F^{1,1}$	$\rightarrow$	$(. E^{1,3})$	$E_{I_0}(EC_1) = \{ (EC_1, -1) + 1 \}$
	$E^{2,1}$	$\rightarrow$	$. E^{2,2} + T$	
	$E^{3,1}$	$\rightarrow$	$. T^{3,2}$	
	$T^{4,1}$	$\rightarrow$	$. T^{4,2} * F$	
	$T^{5,1}$	$\rightarrow$	$. F^{5,2}$	
	$F^{6,1}$	$\rightarrow$	$. (E)$	
	$F^{7,1}$	$\rightarrow$	$. i^{7,2}$	
I_5:	F	$\rightarrow$	$i .$	
I_6:	E	$\rightarrow$	$E + . T$	$E_{I_0}(EC_1) = \{ (EC_1, -2) \}$
	T	$\rightarrow$	$. T * F$	
	T	$\rightarrow$	$. F$	
	F	$\rightarrow$	$. (E)$	
	F	$\rightarrow$	$. i$	
I_7:	T	$\rightarrow$	$T * . F$	$E_{I_0}(EC_l) = \{ (EC_1, -2) \}$
	F	$\rightarrow$	$. (E)$	
	F	$\rightarrow$	$. i$	

I_8: F → (E .)

E → E . + T

I_9: E → E + T .

T → T . * F

I_{10}: T → T * F .

I_{11}: F → (E) .

(a) (b)

Superscripts are for discriminating occurrences of grammar symbols in later explanations.

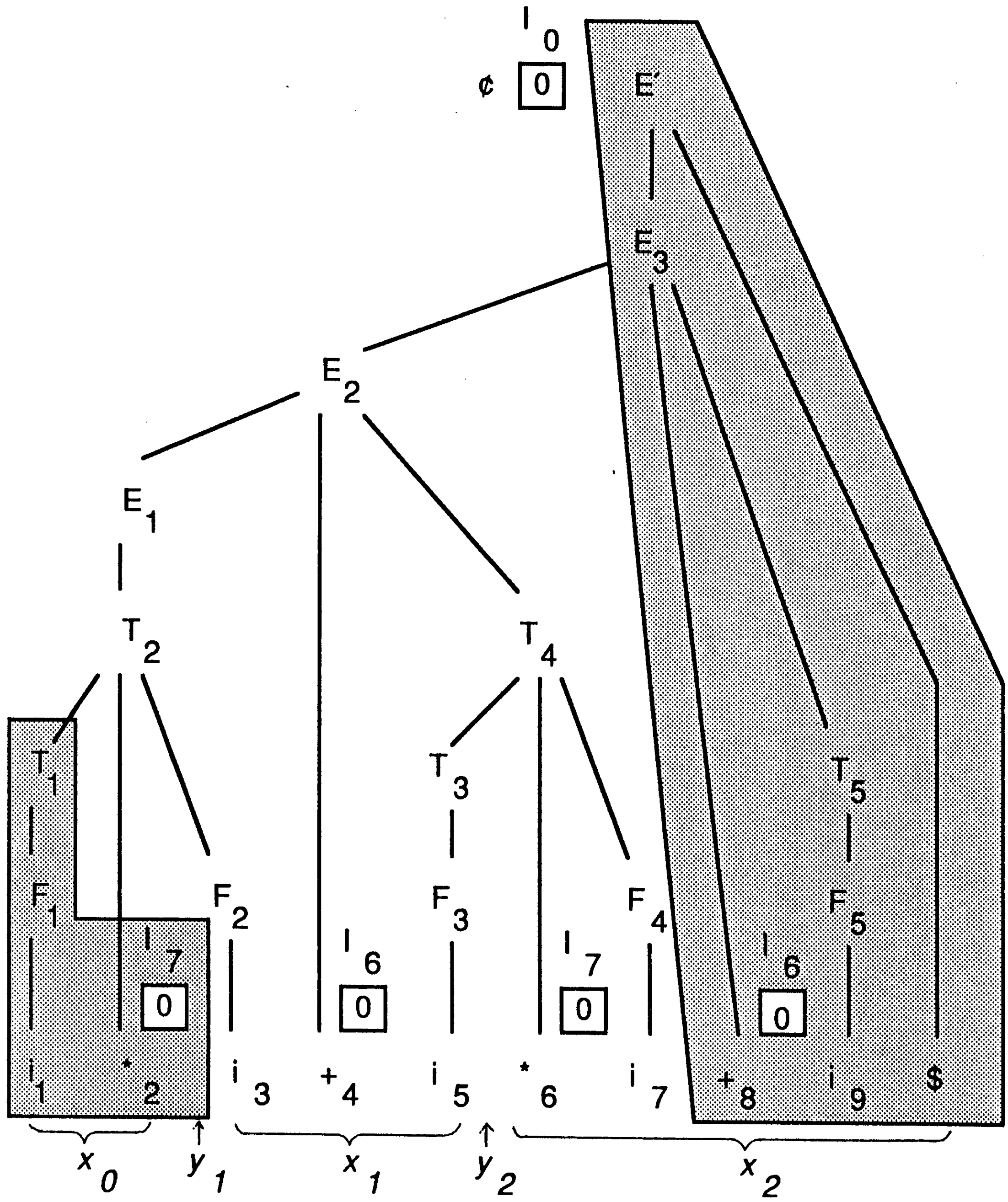
(a) LR states for gramar G1 (canonical LR(0) collection)

(b) semantic expressions corresponding to each LR state

Fig. 3 LR states and semantic expressions for grammar G1 and AG1

STATE	action						goto		
	i	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

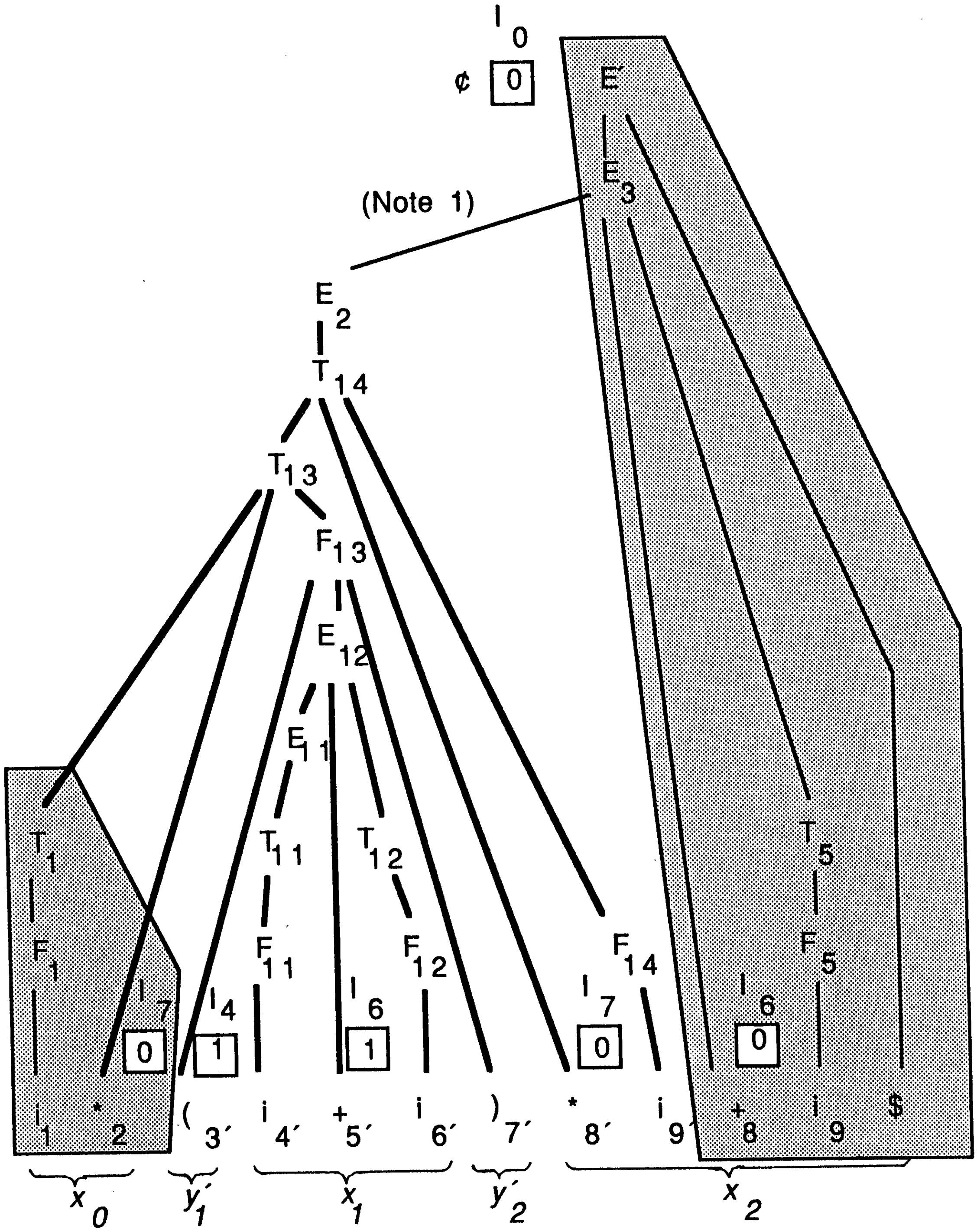
Fig. 4 Parsing table for grammar G1



I_i and the associated box mean an LR state and IN(I_i), i.e. inherited attributes evaluated at LR state I_i, respectively.
F_1, F_3 and F_5 may be omitted as unit production.

(a) original parse tree (ignore I_i etc.)
(b) original APT (with I_i etc.)

Fig. 5 Original parse tree and APT



——— original parse tree   ━━━ new parse subtree

F_1, F_{11}, T_{11}, F_{12} and F_5 may be omitted as unit production.

(Note 1) cut the original arc and establish a new arc.

(a) modified parse tree (ignore I_i etc.)

(b) modified APT (with I_i etc.)

Fig. 6 Modified parse tree and APT

parse stack	input
0 T 2 * 7	(i + i) * i + i $
0 T 2 * 7 (4	i + i) * i + i $
0 T 2 * 7 (4 i 5	+ i) * i + i $
0 T 2 * 7 (4 F 3	+ i) * i + i $
0 T 2 * 7 (4 T 2	+ i) * i + i $
0 T 2 * 7 (4 E 8	+ i) * i + i $
0 T 2 * 7 (4 E 8 + 6	i) * i + i $
0 T 2 * 7 (4 E 8 + 6 i 5	) * i + i $
0 T 2 * 7 (4 E 8 + 6 F 3	) * i + i $
0 T 2 * 7 (4 E 8 + 6 T 9	) * i + i $
0 T 2 * 7 (4 E 8	) * i + i $
0 T 2 * 7 (4 E 8) 11	* i + i $
0 T 2 * 7 F 10	* i + i $
0 T 2	* i + i $
0 T 2 * 7	i + i $
0 T 2 * 7 i 5	+ i $
0 T 2 * 7 F 10	+ i $
0 T 2	+ i $
0 E 1	+ i $

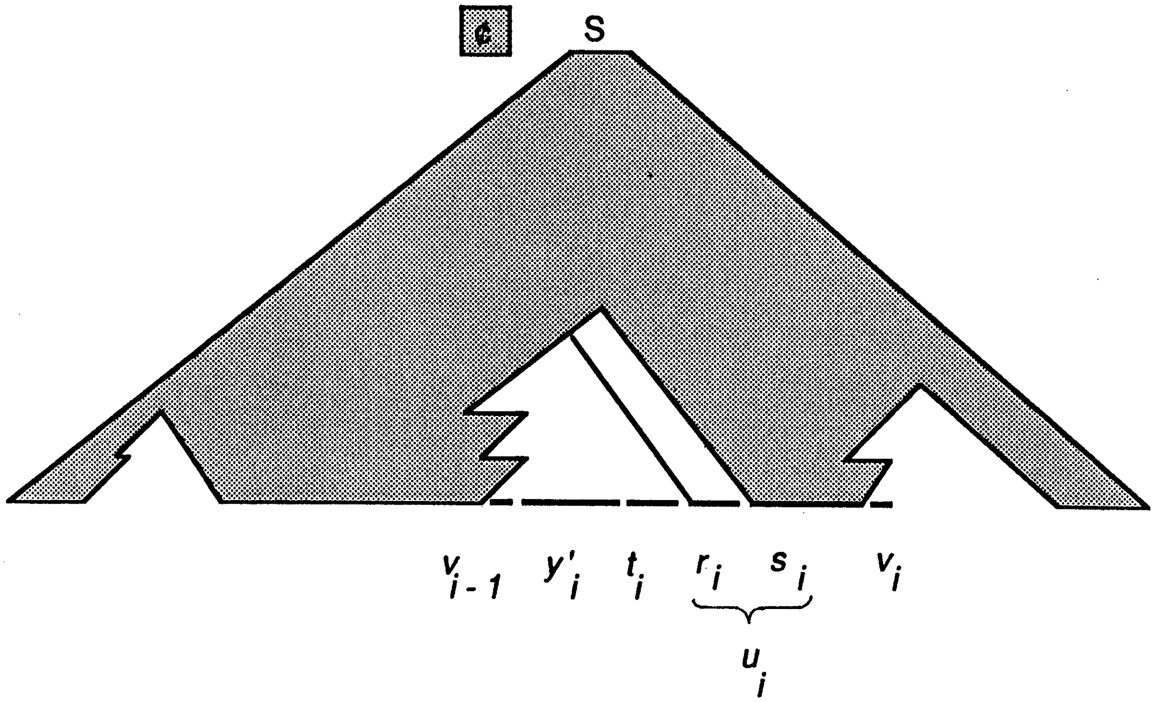
(nil,nil) at the bottom of the parse stack is omitted.
(X, p_X) and I_i in the parse stack are just written as X and i, respectively.

Fig. 7 Incremental parsing of the modified input i * (i + i) * i + i

parse stack	input	evaluation
0	i * i + i * i + i $	$IN(I_0)=\{0\} \rightarrow ¢$
0 i 5	* i + i * i + i $	
0 F 3	* i + i * i + i $	
0 T 2	* i + i * i + i $	
0 T 2 * 7	i + i * i + i $	$IN(I_7)=\{0\} \rightarrow *$
0 T 2 * 7 i 5	+ i * i + i $	
0 T 2 * 7 F 10	+ i * i + i $	
0 T 2	+ i * i + i $	
0 E 1	+ i * i + i $	
0 E 1 + 6	i * i + i $	$IN(I_6)=\{0\} \rightarrow +$
0 E 1 + 6 i 5	* i + i $	
0 E 1 + 6 F 3	* i + i $	
0 E 1 + 6 T 9	* i + i $	
0 E 1 + 6 T 9 * 7	i + i $	$IN(I_7)=\{0\} \rightarrow *$
0 E 1 + 6 T 9 * 7 i 5	+ i $	
0 E 1 + 6 T 9 * 7 F 10	+ i $	
0 E 1 + 6 T 9	+ i $	
0 E 1	+ i $	
0 E 1 + 6	i $	$IN(I_6)=\{0\} \rightarrow +$
0 E 1 + 6 i 5	$	
0 E 1 + 6 F 3	$	
0 E 1 + 6 T 9	$	
0 E 1	$	

$(nil, p_¢)$ at the bottom of the parse stack is omitted.
(X, p_X) and I_i in the parse stack are just written as X and i, respectively.
$IN(I_i)=\{v,...\} \rightarrow X$ in evaluation means: evaluate $IN(I_i)$ according to semantic expressions, get values v,... for equivalence classes, and store them into node corresponding to X.

Fig. 8 Moves of the normal evaluator for the original input i * i + i * i + i



Shaded part is valid

Fig. 9 Modified APT

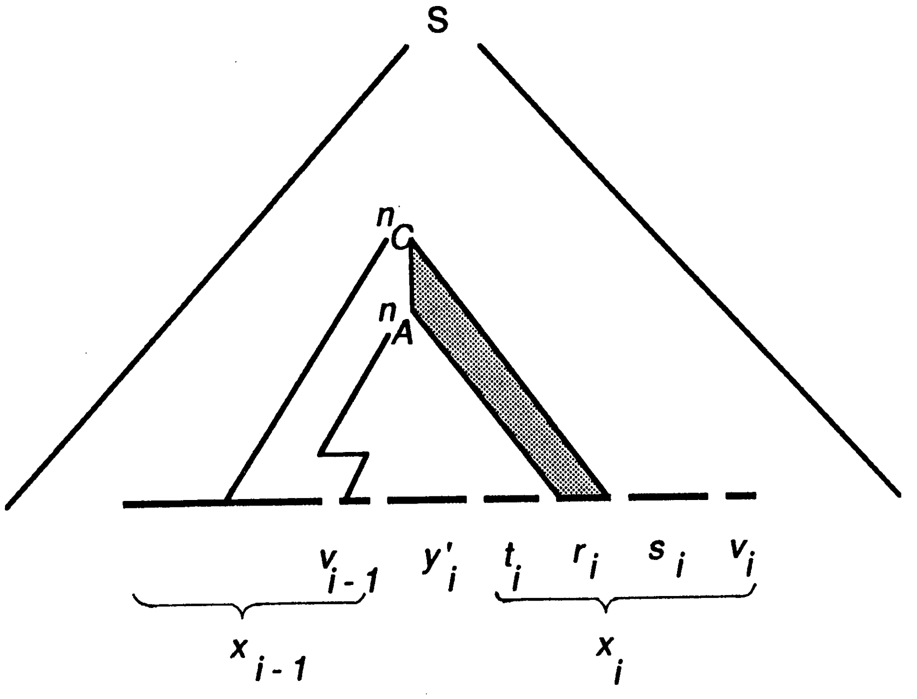


Fig. 10 Re-evaluation of attribute values

parse stack	input	evaluation
0 T 2 * 7	(i + i) * i + i $	
0 T 2 * 7 (4	i + i) * i + i $	$IN(I_4)=\{0\} \rightarrow$ (
0 T 2 * 7 (4 i 5	+ i) * i + i $	
0 T 2 * 7 (4 F 3	+ i) * i + i $	
0 T 2 * 7 (4 T 2	+ i) * i + i $	
0 T 2 * 7 (4 E 8	+ i) * i + i $	
0 T 2 * 7 (4 E 8 + 6	i) * i + i $	$IN(I_6)=\{1\} \rightarrow$ +
0 T 2 * 7 (4 E 8 + 6 i 5	) * i + i $	
0 T 2 * 7 (4 E 8 + 6 F 3	) * i + i $	
0 T 2 * 7 (4 E 8 + 6 T 9	) * i + i $	
0 T 2 * 7 (4 E 8	) * i + i $	
0 T 2 * 7 (4 E 8) 11	* i + i $	
0 T 2 * 7 F 10	* i + i $	
0 T 2	* i + i $	
0 T 2 * 7	i + i $	$IN(I_7)=\{0\} \rightarrow$ *
0 T 2 * 7 i 5	+ i $	
0 T 2 * 7 F 10	+ i $	
0 T 2	+ i $	
0 E 1	+ i $	

Fig. 11 Incremental evaluation of the modified input i * (i + i) * i + i

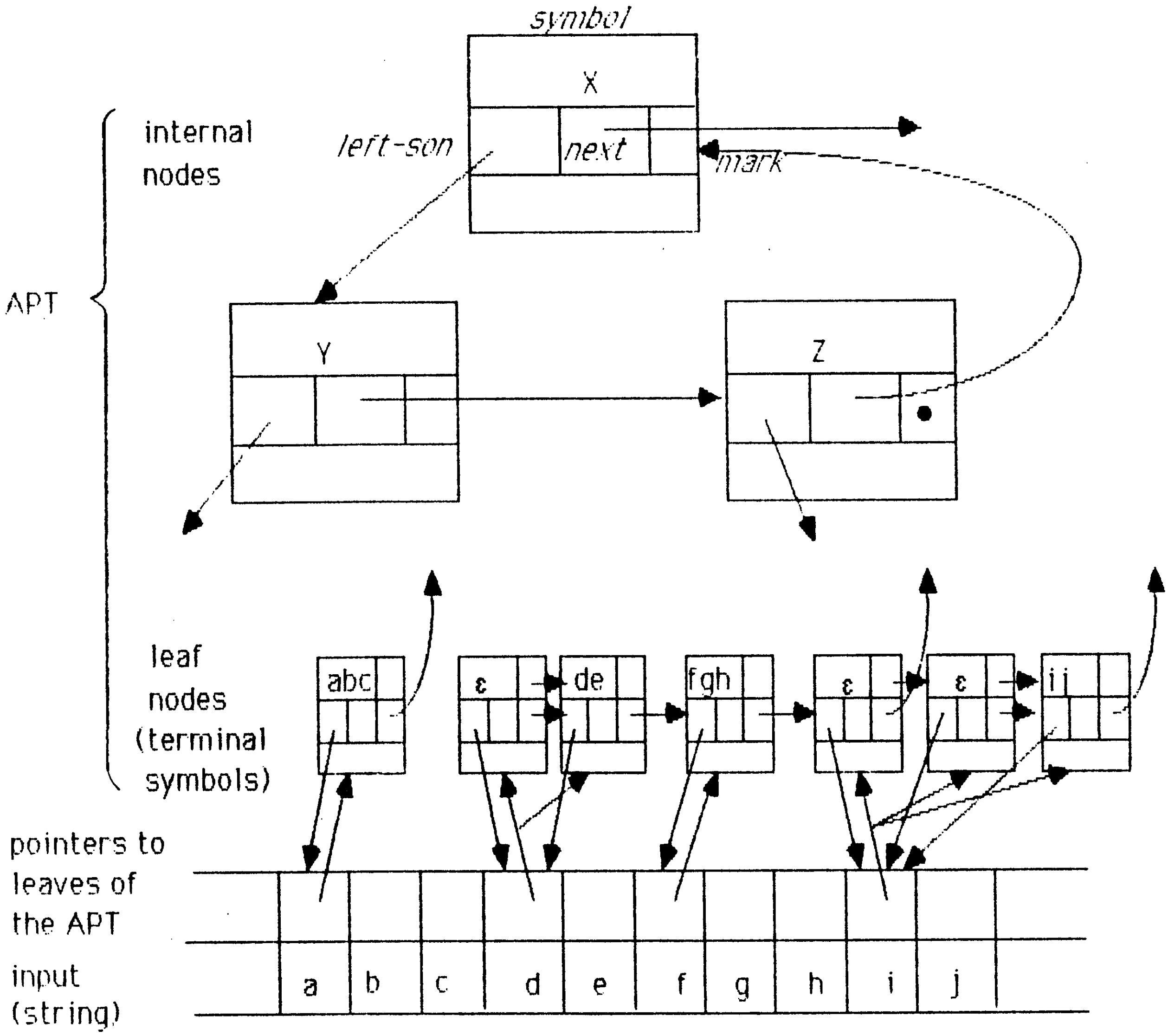


Fig. A1 Possible data structure

INSTITUTE OF INFORMATION SCIENCES AND ELECTRONICS
UNIVERSITY OF TSUKUBA
TSUKUBA-SHI, IBARAKI 305 JAPAN

REPORT DOCUMENTATION PAGE	REPORT NUMBER ISE-TR-88-66

TITLE

Incremental Attribute Evaluation and Parsing
Based on ECLR-attributed Grammars

AUTHOR(S)

Masataka Sassa

REPORT DATE	NUMBER OF PAGES
March 19, 1988	34 + figures
MAIN CATEGORY	**CR CATEGORIES**
Programming Languages - Processors	D.3.4, D.3.1, D.2.3, D.2.6

KEY WORDS

Incremental attribute evaluation, One-pass attribute grammars, ECLR-attributed grammars, Incremental Parsing, LR parsing

ABSTRACT

A method of incremental attribute evaluation and parsing is described. It is based on a class of one-pass attribute grammars called ELCR-attributed grammars which works with LR parsing. The method unifies incremental attribute evaluation and incremental parsing in a single algorithm. It is expected to be space efficient with respect to inherited attributes. Multiple substitutions in the original input are also allowed.

SUPPLEMENTARY NOTES

The report is also published as Report A-1988-9 from Dept. of Computer Science, Univ.of Helsinki.