数理解析研究所講究録
586巻 1986年 44-64

Optimization of Attribute Evaluation in ECLR-attributed Grammars

Harushi Ishizuka[+], Masataka Sassa[++]
and Ikuo Nakata[++]
(石塚 治志 佐々 政孝 中田 育男)

[+] Doctoral Program in Engineering
[++] Inst. of Inf. Sciences and Electronics
Univ. of Tsukuba

Abstract

ECLR-attributed grammars belong to a class of grammars called LR-attributed grammars, for which attributes can be evaluated in a single pass during LR-parsing. A feature of ECLR-attributed grammars is that they handle several inherited attributes as an equivalence class. This makes attribute evaluation more efficient than that of the original LR-attributed grammars.

In this paper, we present a method for the optimization of attribute evaluation in ECLR-attributed grammars. By this optimization, evaluations of the "copy rule", where the value of an equivalence class is copied to another area for that equivalence class, will be eliminated. This can be realized at generation time by solving a certain data flow equation based on the relation of derivation between LR-items.

In general, there are many copy rules found in attribute grammars. Therefore, it is expected that the efficiency of attribute evaluation will be improved sufficiently by this optimization.

1. Introduction

ECLR-attributed grammars (ECLR-AGs) [Sas] belong to a class of grammars called LR-attributed grammars (LR-AGs) [Jon] for which attributes can be evaluated in a single pass during LR-parsing. A feature of ECLR-AGs is that several inherited attributes which have the same value are grouped together and handled as an equivalence class. This makes attribute evaluation more efficient (mainly in space) than that of the original LR-AGs.

We have implemented a compiler generator, called Rie [Ish], based on ECLR-AGs and have written some compilers using it. The method of attribute evaluation in Rie, however, was not optimal, because it does not eliminate the evaluation of "copy rules", where the value of an equivalence class is copied to another area for that equivalence class, which occur rather frequently.

In this paper, we present a method of attribute evaluation which is optimized by eliminating evaluations of the copy rules above.

Optimizations of this kind have been discussed for L-attributed grammars under LL-parsing [Kos]. In the case of LR-parsing, however, it has been considered rather difficult to apply this optimization. The reason for this is that the LR-state can not determine the current syntax uniquely. Nevertheless, the method described here can be realized at generation time merely by solving a certain data flow equation based on the relation of derivation between LR-items.

It is well known that there are many copy rules in attribute grammars. For the PL/0 compiler we have written in ECLR-AGs on Rie for example, 80% of the semantic rules for inherited attributes were copy rules. Therefore, it is expected that the efficiency of attribute evaluation will be improved sufficiently by this optimization and ECLR-AGs will be a more practical class of attribute grammars.

2. Preliminalies

Here, we give some preliminary definitions and an informal explanation for the class of ECLR-attributed grammars.

In the following, an attribute a of symbol X is represented by X.a. The set of inherited attributes and synthesized attributes of symbol X are represented by AI(X) and AS(X), respectively.

We restrict our attention to classes of attribute grammars for which attributes can be evaluated in a single pass during parsing (from left to right) without making a syntax tree. Such classes are called L-attributed grammars and are defined below. In the definition we assume the following.

Assumption 2.1

Semantic rules are in Bochmann normal form [Cou], that is, for the production $X_0 \rightarrow X_1 \ldots X_n$, only attributes in $AI(X_0) \cup AS(X_i)$ $(1 \le i \le n)$ can appear in the right side of semantic rules.

Def 2.1 (L-attributed)

An attribute grammar is L-attributed iff for any production $X_0 \rightarrow X_1 \ldots X_n$ the following conditions hold:

(1) the attribute occurrences in $AI(X_k)$ $(1 \le k \le n)$ depend only on the values of attribute occurrences in $AI(X_0) \cup AS(X_i)$ $(1 \le i \le k-1)$

(2) the attribute occurrences in $AS(X_0)$ depend only on the values of attribute occurrences in $AI(X_0) \cup AS(X_i)$ $(1 \le i \le n)$.

Those conditions state that the attributes must be computable in a strict left to right order. In an LR-parsing which we concern, however, another condition arises since we can "enter" many productions in parallel. For example, when we enter the following LR-state,

$$\left\{\begin{array}{llll} A & \rightarrow V & \text{'='} & \cdot E^1 \\ E^2 & \rightarrow \cdot E^3 & \text{'+'} & T \\ & \mid \cdot T & & \end{array}\right\}$$

the value of attributes in $AI(E^1)$ and $AI(E^3)$ must coincide. (For simplicity, we often show only the core of the LR-items without lookahead.) This condition for inhetited attributes is called LR-attributed. We give an informal definition of LR-attributed grammars and the more precise one can be found in [Jon][Sas'].

Now, we define a set of inherited attributes, IN(S), which should be evaluated at the LR-state S.

Def 2.2 (IN)

On a given LR-state S,

$$IN(S) = \left\{ B.b \;\middle|\; \begin{array}{l} B.b \in AI(B),\ B \text{ is a nonterminal} \\ \text{such that } [A\text{->}\alpha \cdot B\ \beta] \text{ is an LR-item of } S \end{array} \right\}$$

An LR-attributed grammar (LR-AG) is defined as follows.

Def 2.3 (LR-attributed)

An attribute grammar G is LR-attributed iff

(1) G is L-attributed

(2) for any LR-state S of G, and for any inherited attribute A.a ∈ IN(S), the semantic expression [Sas] for A.a can be uniquely determined.

In the definition of LR-AGs, each inherited attribute is assumed to be handled separately. But it often happens at an LR-state S that the values of several inherited attributes in IN(S) are identical. We have defined a class of attribute grammars named ECLR-attributed grammars (ECLR-AGs) by taking this into account. In ECLR-AGs, several inherited attributes which have the same value at any LR-state in which they appear are grouped together and handled as an equivalence class in contrast with LR-AGs.

Def 2.4 (equivalence class)

Let X.a ∈ AI(X) and X'.a' ∈ AI(X'). If the semantic expressions of X.a and X'.a' are identical for any LR-state S such that {X.a, X'.a'} ∈ IN(S), we say that X.a and X'.a' belong to the same equivalence class. (Note that this definition also allows an equivalence class which has only one inherited attribute.)

We assume that all inherited attributes of a given grammar are partitioned into a set of equivalence classes, EC_i $(1 \leq i \leq n)$, such that $EC_i \cap EC_j = \phi$ $(i \neq j, 1 \leq i, j \leq n)$ and $EC_i \neq \phi$ $(1 \leq i \leq n)$.

Def 2.5 (ECLR-attributed)

An attribute grammar G is ECLR-attributed with respect to a set of equivalence classes, EC_i $(1 \leq i \leq n)$, iff

(1) G is L-attributed

(2) for any LR-state S of G, and for any inherited attribute $A.a \in IN(S) \cap EC_i$, the semantic expressions for A.a's can be uniquely determined and are identical.

To handle several inherited attributes as an equivalence class makes attribute evaluation more efficient (mainly in space) than that of the original LR-AGs. The formal definition of ECLR-AGs is found in [Sas].

3. Outline of the optimization

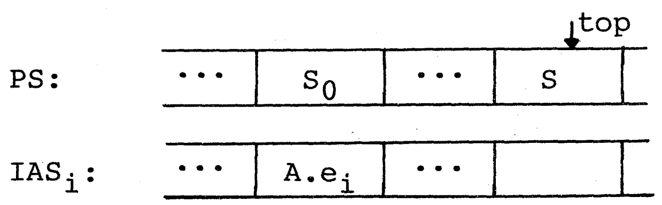
Here, we will give an outline of the optimized attribute evaluation dividing it into three sections, namely, a naive one, an optimized one in a simple case and one in a complicated case.

3.1 Naive attribute evaluation

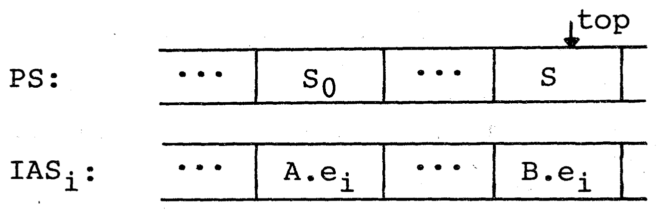
Before presenting the method of the optimized attribute evaluation, we will briefly look at the method of attribute evaluation used in the compiler generator Rie as a naive version of attribute evaluation in ECLR-AGs [Sas].

The evaluation of inherited attributes in ECLR-AGs is done using two kinds of stacks. In addition to the usual parsing stack 'PS' for LR-parsing, attribute stacks which behave synchronously with the parsing stack are used. The stacks 'IAS_i' are for storing inherited attributes, each corresponding to the i-th equivalence class of inherited attributes (denoted by EC_i), and the stack 'SAS', not explained here, is for synthesized attributes.

The set of inherited attributes belonging to an equivalence class is evaluated at the time when the LR-parser goes to a new LR-state. (Henceforth we may often identify an equivalence class with the inherited attributes belonging to that equivalence class.) Its value is unique in ECLR-AGs and it is stored into the appropriate attribute stack (IAS_i). More precisely, at the LR-state $S \ni \{A \rightarrow \alpha \cdot B \beta\}$, the value of inherited attributes of B belonging to an equivalence class EC_i ($B.e_i$) is computed and stored into IAS_i. Now, let the semantic rule for $B.e_i$ be of the form "$B.e_i := A.e_i$". Before computing the value of $B.e_i$, the stack configuration is as shown below.



Here, the LR-state S_0 is such that $S_0 \ni \{X \rightarrow \gamma \cdot A \delta,\ A \rightarrow \cdot \alpha B \beta\}$ and the value of $A.e_i$ is stored into the location corresponding to S_0, that is at $IAS_i[top-|\alpha|]$. ($|\alpha|$ means the length of α.) So, the value of $B.e_i$ can be computed by the assignment "$IAS_i[top] := IAS_i[top-|\alpha|]$". After the evaluation, the stack configuration becomes as below.



Using this method, it is possible to evaluate all attributes for any ECLR-AG. But as shown in the above example, even a copy rule such as "$B.e_i := A.e_i$" is always evaluated unnecessarily. So, it can easily be seen that there remains much possibility for optimizing the evaluation.

3.2 Optimized attribute evaluation - simple case -

Now, we present a simple case of optimized attribute evaluation.

The optimized attribute evaluation method also uses the parsing stack and the attribute stacks. The main difference from the naive method lies in how to evaluate inherited attributes. The strategies for the evaluation are the following.

(1) Push the value of an equivalence class onto the attribute stack only when the value is defined by a semantic rule which is not a copy rule.

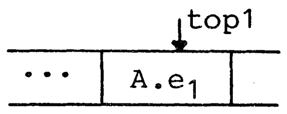
(2) Pop the value of an equivalence class from the attribute stack as soon as possible.

As a result, attribute stacks behave asynchronously with the parsing stack.

Let us consider the following (partial) grammar G1.

```
G1 :  A -> $\alpha$ B $\beta$
      {
        (case 1)  $B.e_1$ := $A.e_1$        /* copy rule */
        (case 2)  $B.e_1$ := f($A.e_1$)     /* non-copy rule */
      }
```

In G1, $B.e_1$ ($A.e_1$) represents an inherited attribute of B (A) in the equivalence class EC_1. During the analysis of G1, the LR-parser goes to an LR-state S_1 such that $S_1 \ni \{A \to \alpha \cdot B \beta\}$. We will see how to compute the value of $B.e_1$ at LR-state S_1 depending on whether the semantic rule for $B.e_1$ is a copy rule or not. It is assumed that the value of $A.e_1$, which is an inherited attribute of the left side symbol, is stored on the top of the attribute stack IAS_1 as illustrated below.

IAS_1: 

(1)

(case 1) <u>copy rule</u> ($B.e_1$:= $A.e_1$)

In this case, we can use the value of $A.e_1$, which has already been stored on the top of IAS_1, instead of the value of $B.e_1$. So there is no need for computing nor handling the attribute stack.

(case 2) <u>non-copy rule</u> ($B.e_1 := f(A.e_1)$)

Unlike (case 1), we must compute the value of $B.e_1$ by evaluating $f(A.e_1)$ and push the obtained value onto IAS_1 (2).

$$IAS_1: \quad \begin{array}{|c|c|c|c} \hline \cdots & A.e_1 & B.e_1 \ \overset{\text{top1}}{\downarrow} & \\ \hline \end{array} \qquad (2)$$

After reading the sentential form corresponding to B, the parser goes to the LR-state S_2 such that $S_2=GOTO(S_1,B) \ni \{A\to\alpha B\cdot \beta\}$. Now the value of $B.e_1$ on the top of IAS_1 becomes useless at LR-state S_2 since $B.e_1$ will never be referenced after S_2 because we assumed that the semantic rules are in Bochmann normal form. So, we can pop $B.e_1$ from IAS_1 at S_2. Push and pop operations in this second case are illustrated in Fig 3.1 with the syntax tree.

In this figure, $e_1\downarrow$ represents the push operation for IAS_1 at LR-state $S_1 \ni \{A\to\alpha\cdot B\ \beta\}$ and $\uparrow e_1$ represents the pop operation for IAS_1 at LR-state $S_2=GOTO(S_1,B)$.

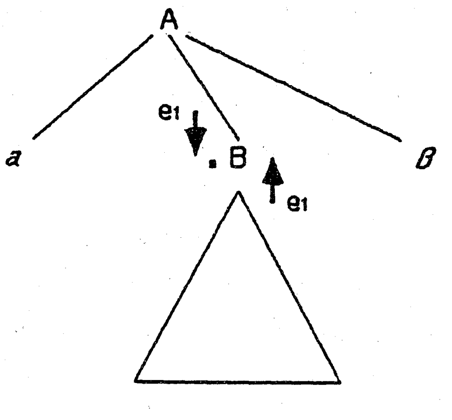


Fig 3.1 Optimized push and pop operation — simple case —

In both (case 1) and (case 2), the stack configuration of IAS_1 just before reading β, i.e. at the LR-state S_2, returns to (1) and the inherited attribute of the left side symbol ($A.e_1$) appears at the top of IAS_1.

Now, we summarize the main idea of this optimization.

While the LR-parser reads the symbol X_i ($1\le i\le n$) which appears in a production rule $X_0 \rightarrow X_1 \ldots X_n$, $X_0.e_k$ which is an inherited attribute of the left side symbol, is being kept on the top of attribute stack IAS_k. This strategy makes it possible to eliminate the evaluation of such copy rules as "$X_i.e_k := X_0.e_k$" ($1 \leq i \leq n$).

3.3 Optimized attribute evaluation - complicated case -

In the previous section, we have discussed the optimized evaluation method in a relatively simple case to help understand it intuitively. Next, we consider two cases which are a little complicated.

The first case involves the push operation. Consider the LR-state and associated semantic rules below.

$$S \ni \left\{ \begin{array}{l} t_1: A \rightarrow \alpha \cdot B \beta \\ t_2: B \rightarrow \cdot X \gamma \\ t_3: X \rightarrow \cdot Y \delta \end{array} \right\} \qquad \begin{array}{l} B.e_1 := f(A.e_1) \\ X.e_1 := B.e_1 \\ Y.e_2 := g(X.e_1) \end{array}$$

We must push e_1 ($B.e_1$, $X.e_1$ which are equal) and e_2 ($Y.e_2$) at the LR-state S, because their values are defined by "non-copy rules". But there is some complexity in determining the point where we can pop e_1 (e_2) corresponding to the push operation for e_1 (e_2).

Because we want to pop attributes as soon as possible, we must identify which LR-item in an LR-state the push operation was done for. Fig 3.2 illustrates the optimal point for the push and pop operations for this case.

As for e_1, we may suppose that push e_1 is done either at t_1 or t_2. However, the value of e_1 may be referenced in the whole subtree rooted at B, because it is not only the value of $X.e_1$ but is also the value of $B.e_1$. So, we can not pop e_1 until we reach the LR-state $GOTO(S,B) \ni \{A \rightarrow \alpha B \cdot \beta\}$ (the point specified by $\uparrow e_1$).

This shows an example of an LR-state which has two or more LR-items for which we must perform the push operation for the same equivalence class e_k (e.g. $e_1\downarrow$, $e_1\Downarrow$). In that case, speaking in terms of the syntax tree, it is appropriate to consider that the push operation for e_k is made at the position whose level is the lowest among them (e.g. $e_1\downarrow$).

On the other hand, push e_2 is done only at t_3 ($e_2\downarrow$), so we can pop e_2 at LR-state GOTO(S,Y) $\ni$ {X->Y· δ} ($\uparrow e_2$). Note that t_3 is the position which has the lowest level with respect to the push operations for e_2. Note also that some push operations are considered to be done at an LR-item in a non-kernel like this LR-item.

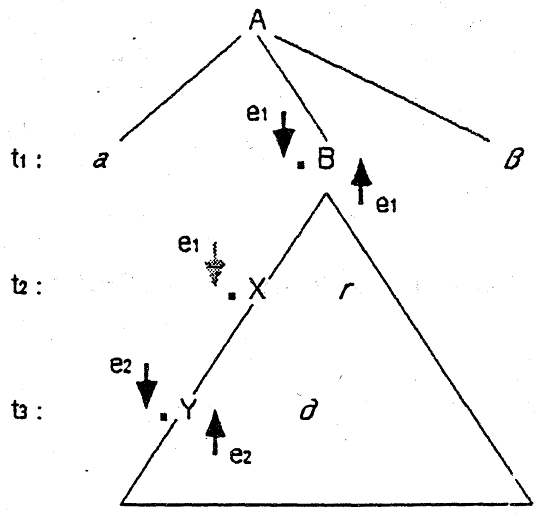


Fig 3.2 Optimized push and pop operation – complicated case (1) –

The second case is more complicated. The complexity here is caused by the fact that an LR-state is unable to determine the syntax uniquely. Consider the LR-state and an associated semantic rule below.

$$S \ni \left\{ \begin{array}{ll} t_1: & A\text{->}\alpha \cdot B\ \beta \\ t_2: & B\text{->}\cdot X\ \gamma \\ t_3: & B\text{->}\cdot Y\ \delta \\ t_4: & X\text{->}\cdot\text{'x'} \end{array} \right\} \qquad Y.e_2 := h(B.e_1)$$

This LR-state shows a situation with two possibilities for the syntax tree as illustrated in Fig 3.3.

Since only $Y.e_2$ must be evaluated at this LR state, the optimal push and pop operation in this case is as illustrated in Fig 3.4.

The push operation for e_2 is meaningful only in case (b) ($e_2\downarrow$). However, we are forced to push e_2 at the LR-state S not only in case (b) but also in case (a), because we can not

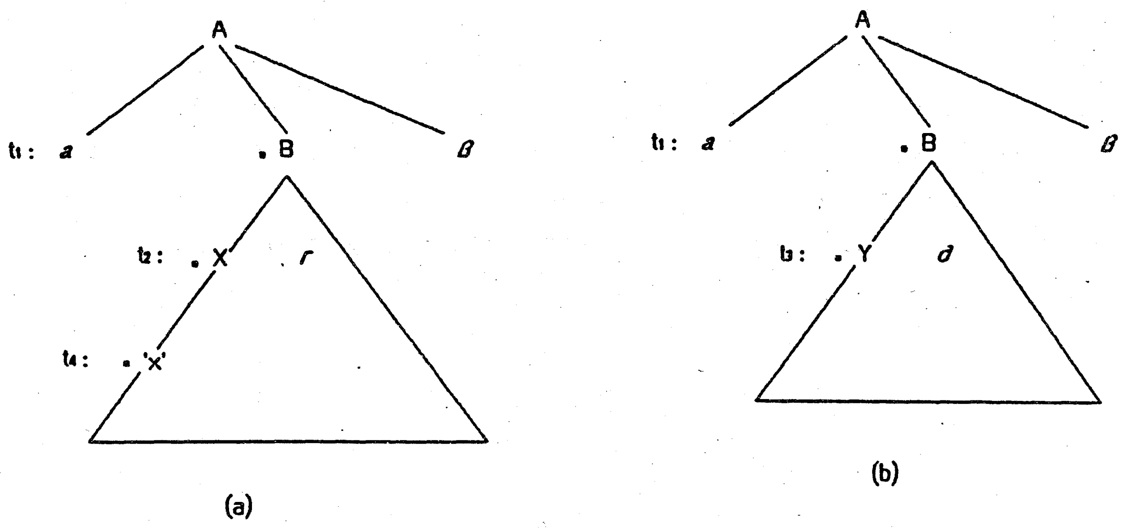


Fig 3.3 Two possible syntax trees in the LR–state

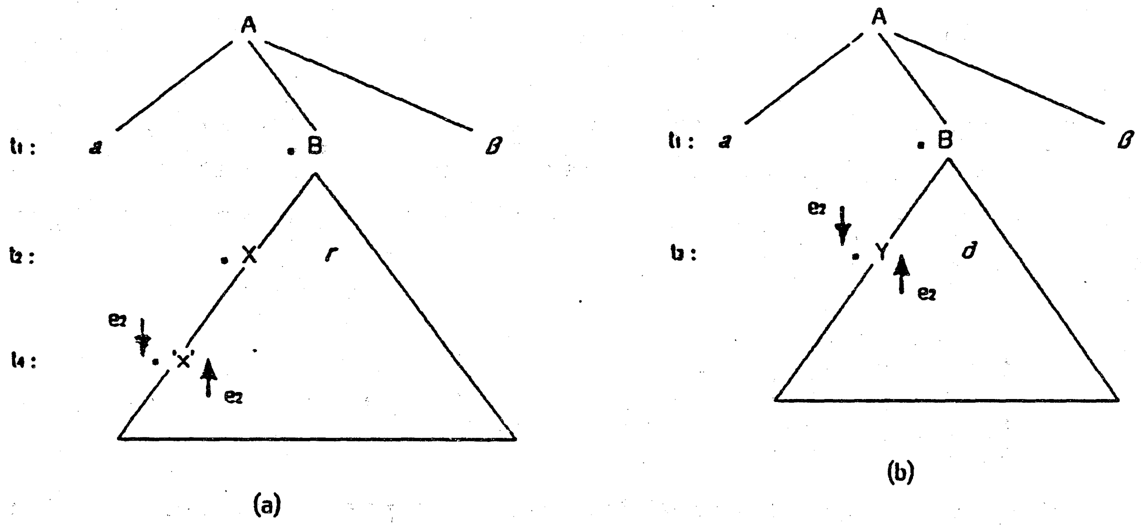


Fig 3.4 Optimized push pop operation – complicated case (2) –

distinguish (a) from (b) only by the LR-state. Let us consider the pop operation under this condition. In case (b), we should pop e_2 at the LR-state GOTO(S,Y) ($\uparrow e_2$) similarly to the first case. In case (a), on the other hand, we want to pop e_2 as soon as the syntax is determined to be (a). So, we pop e_2 at the point just after reading 'x', i.e. GOTO(S,'x') ($\uparrow e_2$). To achieve this, it is proper to suppose that we have pushed e_2 hypothetically at the point specified by $e_2\downarrow$.

Considering these situations, we will formally show in the next section how the optimized attribute evaluation method can be determined at generation time.

4. Realization of the optimized attribute evaluation

To realize the optimized attribute evaluation discussed in the previous section, it is sufficient to compute the following sets of equivalence classes at generation time. For each LR-item t of the form [$A \to \alpha \cdot B \beta$] in an LR-state S, we must compute:

(1) equivalence classes to be actually pushed at t

(2) equivalence classes to be hypothetically pushed at t.

If (1) and (2) are obtained, we can get

(3) equivalence classes to be popped at LR-state GOTO(S,B) $\ni$ {$A \to \alpha B \cdot \beta$}

as the union of (1) and (2).

We will explain the meaning of these sets using the previous example.

Set (1) corresponds to marks ($e_1\downarrow$, $e_2\downarrow$) of Fig 3.2. In this figure, e_1 and e_2 are to be pushed at t_1 and t_2, respectively. Recall that the push operation for an equivalence class is done at the position which has the lowest level with respect to the push operations for that equivalence class in the associated syntax tree.

Set (2) corresponds to mark ($e_2\downarrow$) of Fig 3.4. In this figure, e_2 is actually pushed at t_3 and it is proper to suppose that it is also hypothetically pushed at t_1. Note that t_4 is irrelative to t_3 with respect to the relation of derivation between LR-items.

As we can see from these facts, we must take account of the structure of the syntax tree, or the relation of derivation between LR-items, for computing sets (1) and (2). For this purpose, we use a directed graph, called the <u>State Position Graph</u> (abbreviated as <u>SPG</u>) [Pur], which represents the derivation between LR-items for an LR-state. The definition of the SPG and examples of how it is used are given in the following.

Def 4.1 (SPG)

An SPG is a directed graph of an LR-state whose nodes represent LR-items or special nodes, and whose edges represent direct derivation in the closure of LR-items. Two kinds of special nodes, <u>I</u> and <u>SR</u>, represent an initial node and shift or reduce operations (in parsing), respectively. (For details, see [Pur].)

Ex 4.1

Fig 4.1 and Fig 4.2 show the SPG for the LR-state corresponding to the syntax tree illustrated in Fig 3.2 and Fig 3.4, respectively.

Now, to formalize the computation of sets (1) and (2) at generation time, we introduce three kinds of equivalence class sets, named <u>EVAL</u>, <u>PUSHED</u> and <u>MARK</u>, which are defined for each LR-item.

We will illustrate later, using the previous examples, that these three kinds of sets really meet our requirements.

Def 4.2 (EVAL)

On a given SPG, EVAL(t) is defined as follows for the LR-item t of the form $[A \rightarrow \alpha \cdot B\ \beta]$.

$$\mathrm{EVAL}(t) = \left\{ e_k \;\middle|\; \begin{array}{l} \text{semantic expression (represented only} \\ \text{by known attributes of the kernel) for} \\ B.e_k \text{ evaluated at } t \text{ is a non-copy rule} \end{array} \right\}$$

But, for I and SR, we let EVAL(I) = EVAL(SR) = ϕ.

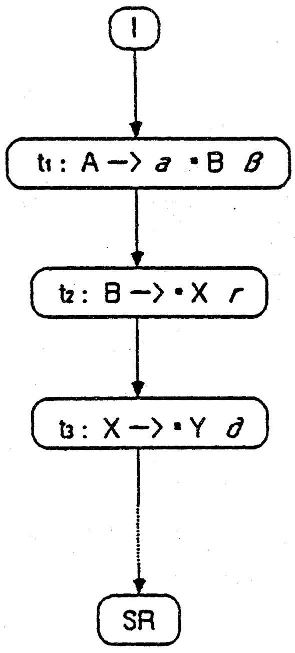


Fig 4.1 SPG of the LR–state corresponding to Fig 3.2

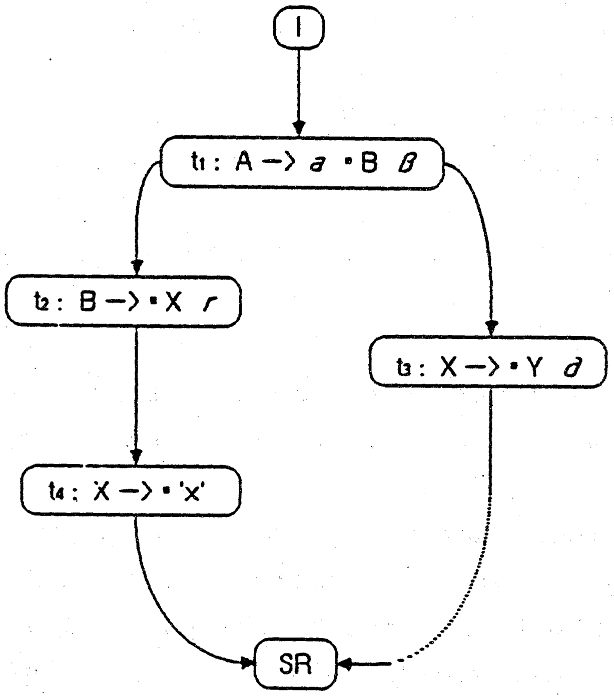


Fig 4.2 SPG of the LR–state corresponding to Fig 3.4

Def 4.3 (PUSHED)

On a given SPG, PUSHED(t) for the LR-item t is defined as the solution of the following data flow equation.

$$\mathrm{PUSHED}(t) = \bigcup_{p \in \mathrm{pred}(t)} [\ \mathrm{PUSHED}(p) \cup \mathrm{EVAL}(p)\ ]$$

Note that PUSHED(t) can be written in another form, that is,

$$\mathrm{PUSHED}(t) = \bigcup_{p \in \mathrm{pred}^{+}(t)} [\ \mathrm{EVAL}(p)\ ]$$

where "pred^{+}" means the non-reflexive transitive closure of "pred". We can see that PUSHED(t) is the set of equivalence classes which are evaluated by non-copy rules at some LR-items which precede t.

The reason why PUSHED is defined as the solution of a data flow equation is that an SPG may contain cycles.

Next, we define MARK as the set of equivalence classes for computing (3).

Def 4.4 (MARK)

On a given SPG, MARK(t) is defined as follows for the LR-item t,

MARK(t) = PUSHED(s) - PUSHED(t)

where s is an arbitrary LR-item such that $s \in \mathrm{succ}(t)$.

We have proved that MARK(t) is unique for any $s \in \mathrm{succ}(t)$.

Ex 4.2

EVAL, PUSHED and MARK for each LR-item on the SPG of Fig 4.1 are shown in Fig 4.3. In this figure, semantic rules are also given as comments.

We have noticed that the actual push operation for e_1 must be done not at t_2 but at t_1, namely, at the position which has the lowest level with respect to the push operations for e_1 in the syntax tree. MARK yields this actual push operation:

$\mathrm{MARK}(t_1) = \mathrm{PUSHED}(t_2) - \mathrm{PUSHED}(t_1) = \{e_1\}$

$\mathrm{MARK}(t_2) = \mathrm{PUSHED}(t_3) - \mathrm{PUSHED}(t_2) = \phi$

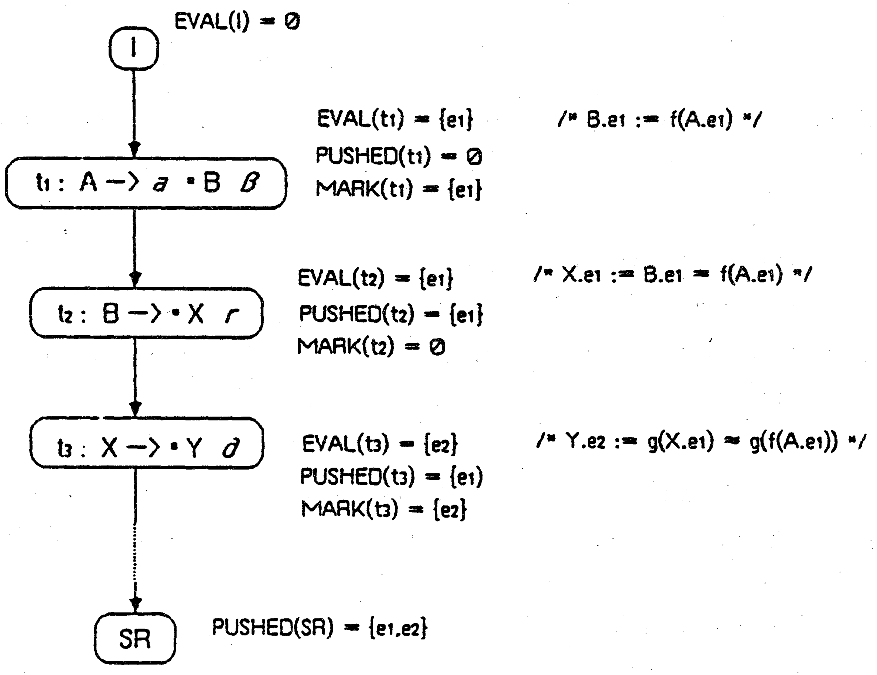


Fig 4.3 EVAL, PUSHED and MARK on the SPG of Fig 4.1

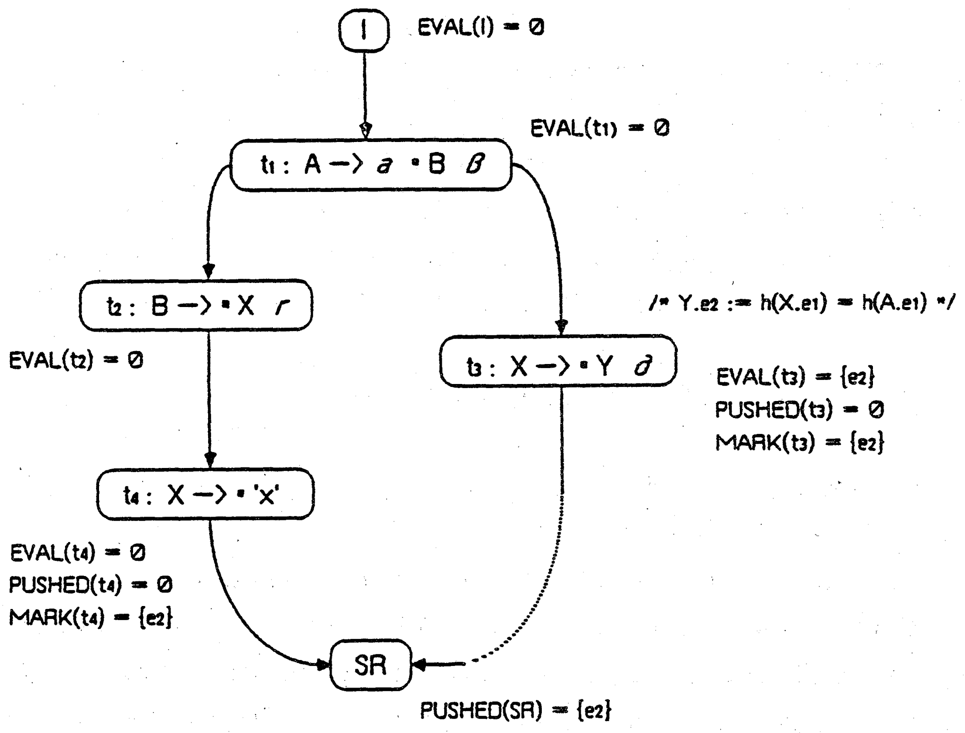


Fig 4.4 EVAL, PUSHED and MARK on the SPG of Fig 4.2

"$e_k \in$ MARK(t)" means "the push operation for e_k is actually done at t".

The reason why MARK meets our requirement is as follows.

The LR-item t which is the actual position with respect to the push operations for e_k must satisfy the following conditions on a given SPG.

(a) $e_k \in$ EVAL(t)

(b) $e_k \notin$ EVAL(p), for each LR-item $p \in$ pred$^+$(t)

Considering that PUSHED(t) = $\bigcup_{p \in \text{pred}^+(t)}$ [EVAL(p)], we can get (a') and (b') from (a) and (b) above:

(a') $e_k \in$ PUSHED(s), for each LR-item $s \in$ succ$^+$(t)

(b') $e_k \notin$ PUSHED(t)

By the definition of MARK, that is MARK(t) = PUSHED(s)-PUSHED(t), we can easily prove:

$e_k \in$ MARK(t), and

$e_k \notin$ MARK(u), for each LR-item $u \in$ {pred$^+$(t) $\cup$ succ$^+$(t)}

Ex 4.3

EVAL, PUSHED and MARK for each LR-item on the SPG of Fig 4.2 are shown in Fig 4.4.

In this case, we have noticed that it is proper to suppose that the push operation for e_2 is also hypothetically done at t_4 so that we can pop e_2 as soon as the syntax is determined to be as in Fig 3.4(a). In Fig 4.4, MARK(t_4) shows us this fact, that is,

$e_2 \in$ MARK(t_4) = PUSHED(SR) - PUSHED(t_4)

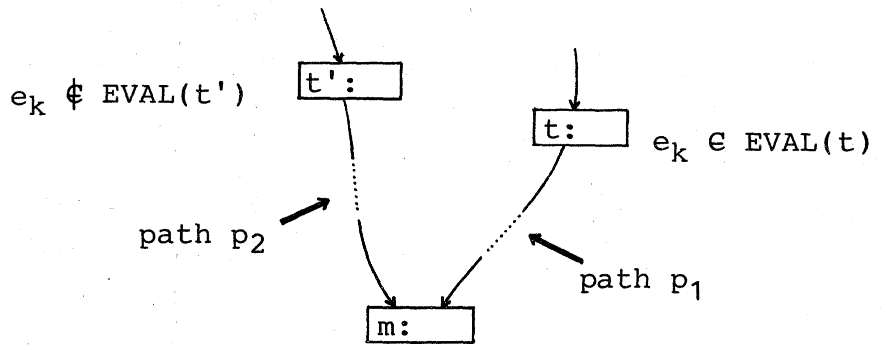
The reason why MARK meets also our requirement in this case is as follows.

In general, if an LR-state shows a situation with two or more possibilities for the syntax tree, there exist two or more paths on the SPG corresponding to those possibilities. Now, let two of such paths be p_1 and p_2. (Here, the word "path" includes the set of LR-items on it.) Since all paths on a given SPG have I as the initial node and SR as the final node, there exists at least one node m such that $m \in p_1 \cap p_2$. (Informally speaking, two paths (p_1 and p_2) merge at m.) Suppose that the actual push operation for e_k occurs only on path p_1. Then, we can say that:

(a) $e_k \in$ EVAL(t), for an LR-item t $\in$ pred$^+$(m) $\cap$ p_1

(b) $e_k \notin$ EVAL(t'), for each LR-item t' $\in$ pred$^+$(m) $\cap$ p_2

This situation is as illustrated below.



So, it is clear that

(a') $e_k \in$ PUSHED(m)

(b') $e_k \notin$ PUSHED(t"), for an LR-item t" $\in$ pred(m) $\cap$ p_2

are true. Hence, we can prove:

$e_k \in$ MARK(t") = PUSHED(m) - PUSHED(t")

This shows that MARK also gives the right position of the hypothetical push operation.

Now, we can conclude that MARK represents set (3) presented at the beginning of this section, namely, for an LR-item t of the form [A->α ·B β] in the LR-state S, if $e_k \in$ MARK(t) then e_k is to be popped at the LR-state GOTO(S,B) $\ni$ {A->α B· β}.

So, for determining the optimized push and pop operations at each LR-state S, we must transfer at generation time the MARK's obtained for LR-items in S to other LR-states succeeding it.

Finally, we present an algorithm performed at generation time by which we can obtain the optimized attribute evaluation.

Algorithm (Determining the optimized push and pop operation)

Input An LR state S augmented by semantic rules and RestPoP's (defined below) of S.

Output Optimized push and pop operations at S and RestPOP's of the LR-states succeeding S.

Variables

PUSHatS: Set of equivalence classes to be pushed at S.

POPatS: Set of equivalence classes to be popped at S.

RestPOP[S,t]: Defined for LR-state S and LR-item t in S. It is used for transferring the candidate set of equivalence classes to be popped from an LR-state to other LR-states succeeding it.

Method

```
(1) Compute POPatS using RestPOP's of kernel(S).
  (a) If kernel(S) contains only one LR-item $k_1$ then
        POPatS := RestPOP[S,$k_1$],
        RestPOP[S,$k_1$] := $\phi$.
  (b) If kernel(S) contains two or more LR-items {$k_1$,···,$k_m$} then
        POPatS := $\phi$. (cf. Note)
      But if $k_i$ is of the form [A->$\alpha$·] then
        "The set of equivalence classes to be popped with that
        reduction" := RestPOP[S,$k_i$].
(2) Make SPG of S and compute EVAL, PUSHED and MARK according to
    Def 4.1-4.4.
(3) PUSHatS := PUSHED(SR).
(4) Transfer to each LR-state which directly succeeds S the
    candidate set of equivalence classes for which pop operations
    may occur at those LR-states. Let t and t' be LR-items of
    the form t:[A->$\alpha$·B$\beta$] $\in$ S and t':[A->$\alpha$B·$\beta$] $\in$ S'=GOTO(S,B),
    respectively.
  (a) If t $\in$ kernel(S) then
        RestPOP[S',t'] := RestPOP[S,t] $\cup$ MARK(t).
  (b) If t $\notin$ kernel(S) then
        RestPOP[S',t'] := MARK(t).
```

Note

This means a delay of the pop operation. Let the LR-state be

$$S \ni \left\{ \begin{array}{l} k_1 : A \to \alpha B \cdot \beta \\ k_2 : B \to B \cdot \gamma \end{array} \right\}$$

and suppose RestPOP[S,k_1]={e_1}, RestPOP[S,k_2]=ϕ. If the syntax (of the input string) corresponds to k_1, it is proper to pop e_1 at S according to RestPOP[S,k_1]. But if the syntax is k_2, it is proper to pop nothing according to RestPOP[S,k_2].

We can not resolve this conflict, because it is impossible to distinguish k_1 from k_2 only by this LR-state. Therefore, we are forced to delay the pop operation at k_1 to the LR-state whose kernel contains only one LR-item (case 1-a), that is, where we can identify the syntax uniquely.

5. Concluding remarks

An optimization of attribute evaluation in ECLR-attributed grammars was given. By this optimization, we can eliminate the evaluation of "copy rules" for an equivalence class. This can be realized at generation time by solving a certain data flow equation based on the relation of derivation between LR-items.

We have not yet measured the effect of this optimization in practice. However, in the case of a PL/0 compiler written using ECLR-attributed grammars, we have obtained the statistical data shown in Table 5.1. Here, 80% of the attribute evaluations for equivalence classes were that of copy rules. This leads us to believe that this optimization will greatly improve the efficiency of attribute evaluation.

As a future problem, it is necessary to implement this optimization on our compiler generator Rie, and to evaluate its efficiency in practice.

source program	41 lines 4 procs
transitions of LR–states (A) with evaluations for equivalence classes (B)	794 times (89 states) 146 times (22 states) (B/A=18.4%)
evaluations for equivalence classes (C) copy rules (D) non–copy rules (E)	150 times 120 times (D/C=80.0%) 30 times (E/C=20.0%)

Table 5.1 Statistics for a compilation of the PL/0 compiler written in ecLR–AGs

Acknowledgements We would like to thank David Duncan for polishing up the English of this paper.

References

[Cou] Courcelle,B., Attribute Grammars: Definitions, Analysis of Dependencies, Proof Methods, in (Lorho, ed.) Methods and Tools for Compiler Construction - An Advanced Course, (Cambridge Univ. Press, 1984).

[Ish] Ishizuka,H. and Sassa,M., A Compiler Generator based on an Attribute Grammar, Proc. 26th Programming Symposium of IPS Japan, pp.69-80 (1985) (in Japanese).

[Jon] Jones,N.D. and Madsen,M., Attribute-influenced LR Parsing, LNCS 94, pp.393-407 (Springer, 1980).

[Kos] Koskimies,K. and Raiha,K.J., Modelling of Space-efficient One-Pass Translation using Attribute Grammars, Softw. Pract. Exper., 13, 2, pp.119-129 (1983).

[Pur] Purdom,P and Brown,C.A., Semantic Routines and LR(k) Parsers, Acta Inf., 14, pp.299-315 (1980).

[Sas] Sassa,M., Ishizuka,H. and Nakata,I., ECLR-attributed Grammars: A practical Class of LR-attributed Grammars, submitted for publication (1985).

[Sas'] Sassa,M., Ishizuka,H. and Nakata,I., A Contribution to LR-attributed Grammars, J. Inf. Process., 8, 3 (1985).