A Contribution to LR-attributed Grammars

MASATAKA SASSA[†], HARUSHI ISHIZUKA[††] and IKUO NAKATA[†]

This paper concerns attribute grammars suitable for LR parsing. A class of attribute grammars called *LR-attributed* grammars has been proposed by Jones and Madsen as a (virtually) maximum class for which attributes can be evaluated during LR parsing. However, the original definition had some insufficient points, and no algorithm was given for checking the LR-attributed property.

In this paper, we first propose corrections and some improvements for the definition of LR-attributed grammars. Next, we present two practical algorithms for checking the LR-attributed property of a given attribute grammar. This work became the basis of a compiler generator called Rie, which we implemented for a subclass of LR-attributed grammars.

Keywords
Attribute grammars, LR-attributed grammars, LR parsing, compiler generators

1. Introduction

Attribute grammars [9][18] are an extension of context-free grammars which unify syntax and semantics of programming languages. They are becoming widely used in compilers, interpreters and in other fields [14][13] [8].

This paper concerns a class of attribute grammars for which attributes can be evaluated in a single pass during LR parsing (LR(1), LALR(1) and SLR(1) parsing) without making a syntax tree. Such grammars make up a subset of the general attribute grammars for which evaluation is made after making a syntax tree. But they are becoming attractive due to their efficiency and practicality and the fact that most modern programming languages are now designed around the easier one-pass processing technique.

A class of attribute grammars called *LR-attributed* grammars has been proposed by Jones and Madsen as a (virtually) maximum class for which attributes (occurrences) can be evaluated during LR parsing [7]. (Here, we concentrate only on "known attributes" of [7] i.e. the attributes which can be evaluated during LR parsing.) Readers unfamiliar with LR-attributed grammars may refer to Appendix A for an informal introduction. In LR-attributed grammars, a subset of inherited attributes can be used as well as synthesized attributes. However, the original definition of LR-attributed grammars had some errors and weak points. It was rather theoretical, and could not be applied directly to practical compilers. The main problems are:

(1) The definition lacked consideration of attribute stack configuration, yielding an erroneous result in some cases.

(2) No algorithm for checking the LR-attributed property was given.

(3) Evaluation of attributes was assumed to be made in a way that all inherited attributes related to an LR state are evaluated and stored separately, causing space and time inefficiency.

In this paper, we show a solution to points (1) and (2). Point (3) will be discussed in a separate paper [15].

Thus first, the definition of Jones et al. is re-formalized to remedy insufficient points. This is done by incorporating an attribute stack configuration into the definition. Also, LR partial states are used instead of LR states to enlarge the class of LR-attributed grammars.

Secondly, two practical algorithms for checking the LR-attributed property of a grammar are developed. While an algorithm by Purdom and Brown [12] can be regarded as a different formalism for checking the LR-attributed property [3], it is not so suited to practical compiler generators since their algorithm requires a rather complicated graph manipulation. Our algorithms are simple and straightforward.

A practical compiler generator, called Rie, was implemented according to a subclass of the LR-attributed grammar [4] [5] [6]. The content of this paper was obtained in the course of developing this compiler generator.

2. Related works

LR-attributed grammars are related to many other works. Probably the first study in this area was that of Watt concerning bottom-up parsing of Affix grammars [19]. In his method, synthesized and inherited attributes (affixes) of the currently processed symbol are stored at the top of a single attribute stack in a certain order. If

[†]Institute of Information Sciences and Electronics,
[††]Doctoral Program in Engineering,
University of Tsukuba, Sakura-mura, Niihari-gun, Ibaraki-ken, 305, Japan

there is a kind of mismatch in the order among attributes of grammar symbols in the right side of a production, his method requires modification of the given grammar. The grammar should be modified to produce the so-called "head grammar" for construction of the parser. This is done by adding special productions (the right side must be empty and the left side is called a "copy-symbol") whose semantic actions are used for copying the attribute values to the top of the attribute stack so that the order of attributes becomes consistent. This addition of productions affects the underlying context-free grammar, causing a problem of possibly destroying the LR-property of the original grammar.

The above problem was not studied in [19], but it was solved by Purdom and Brown [12] and generalized by Pohlmann [11]. Purdom et al. investigated the positions (in the right side of a production) suitable for calling semantic routines in LR(k) grammars. They presented an algorithm to find such positions by labelling positions in a graph associated with each LR (partial) state. The labelling is based on graph properties such as circularity of paths and predominators. Although their work is presented in a formalism different from Jones et al., their results can be regarded as another definition of LR-attributed grammars [3].

There is also a study by Tarhio for evaluating some inherited attributes during LR parsing [17]. In his method, the so-called "uncle" symbol must be searched for at evaluation time, but the method is rather complicated. Besides, a grammar transformation called "uncle transformation" is usually needed beforehand. The possibility or impossibility of this transformation is also related to the work by Purdom et al.

After all, we can conceive of several definitions of LR-attributed grammars according to the formalism of each work, which may not be exactly the same. (For a comparative study, see [17][7].) Among them, we think the definition by Jones et al. is the most natural and potentially the most useful formalism: no grammar transformation is needed in their definition as opposed to other works; the class of attribute grammars by their definition is virtually the largest class under the condition that parser construction need not be modified [17]. Thus, to make a practical compiler generator, we have started with their work and made improvements and developed practical algorithms for checking the LR-attributed property.

3. Attribute Evaluation during LR-Parsing

In this section, we show an analyzer for LR-attributed grammars, which evaluates attributes during LR-parsing. The presentation roughly follows that of Jones et al. [7], but it is a refinement in that LR *partial states* are used instead of LR states for attribute evaluation. Before that, basic notations and definitions are presented. For other definitions concerning attribute grammars, see, for example, [9] [18] and for LR parsing, see [1].

$\mathrm{ASST} \rightarrow \mathrm{V}:=\mathrm{E}^{1,3}$ (1)
{V.env=ASST.env; $\mathrm{E}^{1,3}$.env=ASST.env}

$\mathrm{E}^{2,0} \rightarrow \mathrm{E}^{2,1}+\mathrm{T}^{2,3}$ (2)
{$\mathrm{E}^{2,1}$.env=$\mathrm{E}^{2,0}$.env; $\mathrm{T}^{2,3}$.env=$\mathrm{E}^{2,0}$.env}

$\mathrm{E}^{3,0} \rightarrow \mathrm{T}^{3,1}$ (3)
{$\mathrm{T}^{3,1}$.env=$\mathrm{E}^{3,0}$.env}

$\mathrm{T}^{4,0} \rightarrow \mathrm{P}^{4,1} ** \mathrm{T}^{4,3}$ (4)
{$\mathrm{P}^{4,1}$.env=$\mathrm{T}^{4,0}$.env; $\mathrm{T}^{4,3}$.env=$\mathrm{T}^{4,0}$.env}

$\mathrm{T}^{5,0} \rightarrow \mathrm{P}^{5,1}$ (5)
{$\mathrm{P}^{5,1}$.env=$\mathrm{T}^{5,0}$.env}

$\mathrm{P}^{6,0} \rightarrow$ name (6)
{**condition** name.tag∈$\mathrm{P}^{6,0}$.env}

$\mathrm{P}^{7,0} \rightarrow (\mathrm{E}^{7,2})$ (7)
{$\mathrm{E}^{7,2}$.env=$\mathrm{P}^{7,0}$.env}

Superscripts distinguish occurrences of grammar symbols in productions.

Fig. 1 Attribute grammar G1 (partial grammar).

In the following, the presentation is done using an example attribute grammar G1 as shown in Fig. 1. (The syntax is taken from [12].) Superscripts may be added to grammar symbols to distinguish their occurrences in productions or LR items. For example in Fig. 1, $\mathrm{E}^{1,3}$ means E at the 3rd position of the 1st production. We consider in principle that all grammar symbols in productions have superscripts. But these superscripts are meaningful only if definitions requiring them are used, otherwise, the superscripts should be neglected.

We use the following convention: upper-case names such as A, B, C, . . ., Z, Z' and ASST are nonterminals; Lower-case names such as a, b, c and name, and operator symbols such as :=, + and ** are terminals; X, Y, . . . are grammar symbols (either nonterminals or terminals); α, β, γ, . . . are strings of grammar symbols.

Semantic rules are enclosed in { and }. An attribute a of symbol X is represented by $X.a$. The set of inherited and synthesized attributes of symbol X are represented by $AI(X)$ and $AS(X)$, respectively.

We will often use the term grammar to denote both the entire attribute grammar and the underlying context-free grammar.

Def 3.1 For the production $X_0 \rightarrow X_1 \ldots X_n$ (where X_0 is a nonterminal), $AI(X_0)$ and $AS(X_j)$ ($j=1, \ldots, n$) are called *input attribute occurrences*.

As in much of the literature, we assume the following.

Assumption 3.1 Only input attribute occurrences appear in the right side of a semantic rule of a given grammar (often called *Bochmann normal form*).

Since we want to evaluate attributes during left-to-right parsing, our method is closely related to the L-attributed property of attribute grammars. L-attributed grammars are defined as follows under Assumption 3.1 (a modification of [18]).

Def. 3.2 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 $AI(X_k)$ $(1 \leq k \leq n)$ depend only on the values of attribute occurrences in $AI(X_0) \cup \cup_{i=1}^{k-1} AS(X_i)$.

(2) The attribute occurrences in $AS(X_0)$ depend only on the values of attribute occurrences in $AI(X_0) \cup \cup_{i=1}^{n} AS(X_i)$.

Ex 3.1 Grammar G1 is L-attributed since each production satisfies conditions (1) and (2) of Def 3.2.

Assumption 3.2 In the following, we assume that attribute grammars are L-attributed.

For a given grammar, LR states can be constructed as usual [1]. We assume that the start symbol of the grammar is Z and the grammar is augmented by the production "$Z' \rightarrow Z$".

Def 3.3 An *LR(1) item* (*LR item* or *item* for short) of a grammar G is $[A \rightarrow \alpha \cdot \beta, a]$ where "$A \rightarrow \alpha\beta$" is a production of G, and a is a terminal symbol. Its first component $[A \rightarrow \alpha \cdot \beta]$ is called a *core*.

Def 3.4 A *closure* of a set of LR items, I, is a set of items defined by the following $CLOSURE(I)$.

1. Every item in I is in $CLOSURE(I)$.
2. If $[A \rightarrow \alpha \cdot B\beta, a]$ is in $CLOSURE(I)$, "$B \rightarrow \gamma$" is a production, and b is in $FIRST(\beta a)$, then add the item $[B \rightarrow \cdot \gamma, b]$ to I, if it is not already there.

Here, it is said that $[A \rightarrow \alpha \cdot B\beta, a]$ *directly derives* $[B \rightarrow \cdot \gamma, b]$. The *derives* relation is the non-reflexive transitive closure of the *directly derives* relation.

Def 3.5 The set of items $GOTO(I, X)$ for a set of items I and a grammar symbol X is defined to be the closure of the set of all items $[A \rightarrow \alpha X \cdot \beta, a]$ such that $[A \rightarrow \alpha \cdot X\beta, a]$ is in I. Here, the set of all items $[A \rightarrow \alpha X \cdot \beta, a]$ is called the *kernel* of $GOTO(I, X)$.

For simplicity, we often show only the core of LR items in the discussion below.

Using *CLOSURE* and *GOTO* operations, the collection of sets of LR items, $\{I_0, I_1, \ldots, I_n\}$, is constructed starting from the initial set of LR items, $I_0 = CLOSURE(\{[Z' \rightarrow \cdot Z, \$]\})$ [1]. The set of LR items I_i thus constructed is called an *LR state* S_i from the viewpoint of an LR automaton. (We often identify S_i with I_i.)

The LR parsing table is constructed as follows.

1. a) If $[A \rightarrow \alpha \cdot a\beta, b]$ is in S_i and $GOTO(S_i, a) = S_j$, then set ACTION$[S_i, a]$ to "shift S_j".
   b) If $[A \rightarrow \alpha \cdot, a]$ is in S_i, then set ACTION $[S_i, a]$ to "reduce by $A \rightarrow \alpha$".
   c) If $[Z' \rightarrow Z \cdot, \$]$ is in S_i, then set ACTION $[S_i, \$]$ to "accept".
2. If $GOTO(S_i, A) = S_j$, then GOTO $[S_i, A] = S_j$.
3. All entries not defined by the above rules are set to "error".

(a)
$$\left\{\begin{array}{ll} \text{ASST} \rightarrow \text{V:=} \cdot \text{E}^{1,3} & (1) \\ \text{E}^{2,0} \rightarrow \cdot\, \text{E}^{2,1} + \text{T}^{2,3} & (2) \\ \text{E}^{3,0} \rightarrow \cdot\, \text{T}^{3,1} & (3) \\ \text{T}^{4,0} \rightarrow \cdot\, \text{P}^{4,1} ** \text{T}^{4,3} & (4) \\ \text{T}^{5,0} \rightarrow \cdot\, \text{P}^{5,1} & (5) \\ \text{P}^{6,0} \rightarrow \cdot\, \text{name} & (6) \\ \text{P}^{7,0} \rightarrow \cdot\, (\text{E}^{7,2}) & (7) \end{array}\right\} \text{(b)}$$

(a) is an LR state (S)
(b) is an LR partial state (PS)
(state=S, lookahead=name)

Fig. 2 An LR state and an LR partial state of G1.

As in [12], we subdivide an LR state into (LR-) partial states according to lookahead terminal symbols.

Def 3.6 [12] The *lookahead set* for item $[A \rightarrow \alpha \cdot \beta, a]$ is the set of k symbol (here, we let $k=1$) prefixes of the strings of terminal symbols that can be derived from βa. The *partial state* of S with lookahead b is the set of items in state S that have b in their lookahead set.

Ex 3.2 The partial state PS of S (Fig. 2(a)) with lookahead 'name' is shown in Fig. 2(b).

In practice, considering LR states instead of partial states will suffice in most cases, but here partial states are used for theoretically widening the class of LR-attributed grammars.

For a partial state PS, we define IN(PS) and IN'(PS) as follows. They are the sets of inherited attributes which should be evaluated at partial state PS.

Def 3.7 Given a partial state PS,

IN(PS)={$B.b$ | $B.b$ is an inherited attribute of a nonterminal B such that $[A \rightarrow \alpha \cdot B\beta, a]$ is an LR item of PS. (α, β may be empty.)}

IN'(PS)={$B'.b$ | $B'.b$ is an inherited attribute of a nonterminal B' such that $[A \rightarrow \alpha \cdot B'\beta, a]$ is an LR item of PS, where superscript t is for discriminating occurrences of grammar symbols in LR items.}

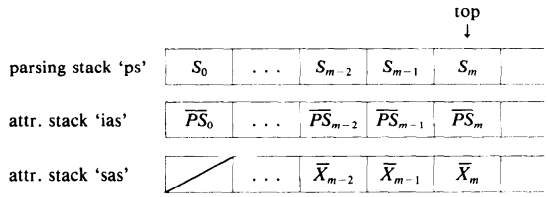
(Here, we consider two versions of productions: in the definition of IN'(PS), superscripts of grammar symbols are taken into account, while in the definition of IN(PS) they are neglected.)

Ex 3.3 IN(PS) and IN'(PS) for the partial state PS of Fig. 2(b) are {E.env, T.env, P.env} and {$\text{E}^{1,3}$.env, $\text{E}^{2,1}$.env, $\text{T}^{3,1}$.env, $\text{P}^{4,1}$.env, $\text{P}^{5,1}$env}, respectively.

The organization of stacks for syntax and semantic analysis of LR-attributed grammars is sketched in Fig. 3. The parsing is made by the usual LR parsing method [1]. In addition to the usual parsing stack 'ps' for LR parsing which contains LR states, two attribute stacks which behave synchronously with the parsing stack are used. The stack 'ias' is for storing inherited attributes and the stack 'sas' is for synthesized attributes.

More formally, let an *attributed parse configuration* (abbreviated as *configuration*) of the analyzer be

$$(S_0 \overline{PS}_0 X_1 \overline{X}_1 \ldots S_{m-1} \overline{PS}_{m-1} X_m \overline{X}_m S_m,\ a_j \ldots a_n \$)$$



S_i is an LR state, $\overline{PS_i}$ is a record containing values of inherited attributes in IN(PS_i), $\overline{X_i}$ is a record containing values of synthesized attributes of X_i.

Fig. 3 Organization of stacks for analysis of LR-attributed grammars.

Here,
— $S_i (i=0, \ldots)$ is an LR state (on the parsing stack 'ps') (S_0 is the initial state $\{[Z' \rightarrow \cdot Z, \$]\}$, where "$Z' \rightarrow Z$" is the first production in the augmented grammar).
— $\overline{PS_i}$ is a record containing values of all inherited attributes in IN(PS_i) (on the stack 'ias') where PS_i is a partial state of S_i with appropriate lookahead. In particular, if a_j is the next input symbol and the current LR state is S_m which is (we only show the core of LR items and omit [and])

$$S_m = \{X_0 \rightarrow \ldots X_{m-1} X_m \cdot X_{m+1} \ldots, X_{m+1} \rightarrow \cdot \ldots, \ldots, X_q \rightarrow \cdot a_j \alpha, \ldots\}$$

then, PS_m ($\subset S_m$) is a partial state of S_m with lookahead a_j

$$PS_m = \{X_0 \rightarrow \ldots X_{m-1} X_m \cdot X_{m+1} \ldots, X_{m+1} \rightarrow \cdot \ldots, \ldots, X_q \rightarrow \cdot a_j \alpha\}.$$

Note that by the nature of LR parsing, $PS_{m-1} \subset S_{m-1}$ is

$$PS_{m-1} = \{X_0 \rightarrow \ldots X_{m-1} \cdot X_m X_{m+1} \ldots, X_m \rightarrow \cdot \ldots, \ldots, X_{q'} \rightarrow \cdot a_{j'} \beta\}$$

and $PS_{m-2} \subset S_{m-2}$ is

$$PS_{m-2} = \{X_0 \rightarrow \ldots \cdot X_{m-1} X_m X_{m+1} \ldots, X_{m-1} \rightarrow \cdot \ldots, \ldots, X_{q''} \rightarrow \cdot a_{j''} \gamma\}.$$

— X_i is a grammar symbol. Note that in practice, grammar symbols X_i's need not be stored as shown in Fig. 3.
— $\overline{X_i}$ is a record containing values of all synthesized attributes in $AS(X_i)$ (on the stack 'sas').
— $a_j \ldots a_n$ is the remaining input.

Since IN(PS_i) may differ from partial state to partial state, the type of an element of the stack 'ias' is the union of records, each record containing attribute values of IN(PS_i). Similarly, the type of an element of the stack 'sas' is the union of records, each record containing attribute values of $AS(X_i)$.

Here, we define the offset of an attribute occurrence as its position from the top in the attribute stack organization of Fig. 3, as follows.

Def 3.8 If the configuration of the analysis is

$$(S_0 \overline{PS_0} X_1 \overline{X_1} \ldots S_{m-1} \overline{PS_{m-1}} X_m \overline{X_m} S_m, a_j \ldots a_n \$)$$

then, the *offset* of an attribute occurrence a is o (<0) if a is a synthesized attribute occurrence contained in X_{m+o}, or a is an inherited attribute occurrence contained in PS_{m+o}.

According to the nature of LR parsing discussed above, this means that if the current state is S_m such that

$$X_0 \rightarrow X_{m-k} \ldots X_m \cdot X_{m+1} \ldots \in S_m$$

then, the offset of an attribute occurrence $a \in AI(X_0)$ is $-(k+1)$, and the offset of an attribute occurrence $b \in AS(X_{m-i})$ $(0 \leq i \leq k)$ is $-i$.

An attribute evaluator for LR-attributed grammars which evaluates attributes during LR parsing is shown below. Synthesized attributes of a nonterminal A are evaluated when the parser reduces α by the production "$A \rightarrow \alpha$". Inherited attributes of a nonterminal B are evaluated when the parser goes to an LR state which contains an LR item $[A \rightarrow \beta \cdot B \gamma]$. Thus, attribute evaluation occurs not only at reduction time but also at state transition time, where grammar symbols of the right side of a production are processed. This is in contrast to conventional bottom-up syntax-directed translation where semantic analysis is made only at reduction time.

The following attribute evaluator is a refinement of [7] in that LR partial states are used instead of LR states for the evaluation of inherited attributes.

---

procedure EVAL {attribute evaluator during LR parsing}
begin
  configuration:=(S_0, $a_1 \ldots a_n$\$);
  loop
    let configuration **be**
      ($S_0 \overline{PS_0} X_1 \overline{X_1} \ldots S_{m-1} \overline{PS_{m-1}} X_m \overline{X_m} S_m$, $a_j \ldots a_n$\$);
    action:=ACTION[S_m, a_j] {ACTION in the parse table};
    if action="accept" or action="error" **then** exit;
    PS_m:=PARTIAL-STATE[S_m, a_j]
      {partial state of S_m with lookahead a_j};
    compute values of inherited attributes in IN(PS_m) {see section 4};
    $\overline{PS_m}$:=record containing the computed values;
    configuration:=(. . . $S_m \overline{PS_m}$, $a_j \ldots a_n$\$);
    {Fig. 3 shows the organization of stacks at this point}
    case action **of**
      "shift S":
        $\bar{a}_j$:=record containing values of synthesized attributes of a_j {from lexical analysis};
        configuration:=(. . . $S_m \overline{PS_m} a_j \bar{a}_j S$, $a_{j+1} \ldots a_n$\$);
      "reduce by $A \rightarrow \alpha$":
        compute values of synthesized attributes of A;
        $\overline{A}$:=record containing the computed

```
            values;
        k:=|α|;
        pop configuration down to
              (. . . S_{m-k} PS_{m-k} , a_j . . . a_n$);
        S:=GOTO[S_{m-k}, A] {GOTO in the parse
            table};
        configuration:=(. . . S_{m-k} PS_{m-k} A AS,
                        a_j . . . a_n$)
    end case
  end loop
end
```

Note 3.1 This attribute evaluator can deal with ε-rules.

Note 3.2 In this attribute evaluator, the computation of $\overline{PS}_m$ (values of inherited attributes in IN(PS_m)) is useless if ACTION[S_m, a_j]="reduce by $A\to\alpha$" with $\alpha\neq\varepsilon$, since in this case $\overline{PS}_m$ will soon be popped up. It is easy to insert a test to bypass the computation of $\overline{PS}_m$ in this case [7].

4. Reformulation of LR-attributed Grammars

LR-attributed grammars are reformulated as follows, with several improvements on [7]: a major improvement is that the organization of attribute stacks is considered in the definition; another refinement is that LR partial states are used instead of LR states for attribute evaluation to enlarge the class of LR-attributed grammars.

In the attribute evaluator EVAL of the previous section, $\overline{PS}_m$, the values of inherited attributes in IN(PS_m), must be computed. In LR-attributed grammars, this can be done by observing that the value of an attribute in IN(PS_m) ultimately comes from the values of attributes in the kernel of PS_m. Thus, we can regard the value of each inherited attribute $A.a$ in IN(PS_m) as a function of the input attribute occurrences of the LR items (or LR item) in the kernel of PS_m. The function is called a *semantic expression* and is represented by $E_{PS}(A.a)$ for the attribute $A.a$.

For example, in the partial state PS (Fig. 2(b)) of grammar G1, the value of E.env which is in IN(PS) can be computed as follows. Since E.env appears in two places, i.e., $E^{1,3}$.env and $E^{2,1}$.env, these two attribute values must be taken into consideration. First, the value of $E^{1,3}$.env is defined to be equal to ASST.env according to the second semantic rule of grammar G1 (Fig. 1). (Here, ASST.env is an input attribute of the (only one) item in the kernel, which is {ASST→V:= · E}.) To get the value of ASST.env from the attribute stacks, we also have to know the position of ASST.env (here the attribute stack is 'ias' since ASST.env is an inherited attribute). Since 'ps[top-1]' corresponds to the LR state {ASST→V · :=E} and 'ps[top-2]' corresponds to the LR state {. . .→. . . · ASST . . . , ASST→ · V:=E, . . .}, we know that ASST.env is at 'top-2' of the attribute stack, i.e., the offset of ASST.env is '−2'. We denote this by {(ASST.env, −2)}. Second, the value of $E^{2,1}$.env is defined to be equal to $E^{2,0}$.env. As the LR item [$E^{2,0}\to$ · $E^{2,1}+T^{2,3}$] is derived from itself or from [ASST→V:= · $E^{1,3}$], the value of $E^{2,0}$.env comes from $E^{1,3}$.env. Its value is equal to ASST.env and the offset of ASST.env is '−2' from the first observation. Putting the two cases together by taking the set union of these cases, we can see that

$$\begin{aligned} &E_{PS}(E.env)\\ &=\{\text{expression of } E^{1,3}.env\}\cup\{\text{expression of } E^{2,1}.env\}\\ &=\{(ASST.env,\ -2)\}. \end{aligned}$$

By making the above inspection more precise, we get the following definitions.

Def 4.1 *Input attributes* of a partial state PS are either
(1) inherited attributes of A such that $[A\to\alpha\cdot\beta, a]$ is in the kernel of PS, or
(2) synthesized attributes of grammar symbols appearing in α such that $[A\to\alpha\cdot\beta, a]$ is in the kernel of PS.

For each inherited attribute $A.a$ in IN(PS), the semantic expression $E_{PS}(A.a)$ is a set of expressions in terms of the input attributes of PS as follows.

Def 4.2 For a partial state PS of a grammar, and for an $A.a\in$IN(PS), $E_{PS}(A.a)$ is defined as a set of expressions as folows.

$$E_{PS}(A.a)=\bigcup_{A'.a\in IN'(PS)} F_{PS}(A'.a)$$

where $A'.a\in$IN'(PS) is the same attribute as $A.a\in$IN'(PS) but with A being superscripted to distinguish occurrences of A in LR items in PS. The domain of F_{PS} is IN'(PS). $F_{PS}(A'.a)$ will be represented in terms of the input attributes of PS. $F_{PS}(A'.a)$ is the smallest solution [1] of the following equations.

(1) If A' occurs in an LR item in the kernel of PS:
(let (the core of) the LR item be $[B\to\alpha\cdot A'\beta]$)

$$\begin{aligned} &F_{PS}(A'.a)\\ &=\{f((B.b_1,\ o_{B.b_1}),\ (B.b_2,\ o_{B.b_2}),\ \ldots,\\ &\qquad (X_1.x_1,\ o_{X_1.x_1}),\ (X_2.x_2,\ o_{X_2.x_2}),\ \ldots)\mid \end{aligned}$$

"$A.a=f(B.b_1, B.b_2, \ldots, X_1.x_1, X_2.x_2, \ldots)$" is the semantic rule for defining $A'.a$, where $B.b_i\in AI(B)$ and $X_j.x_j\in AS(X_j)$ for a symbol X_j in α.
$o_{B.b_i}$ or $o_{X_j.x_j}$ is the offset of $B.b_i$ or $X_j.x_j$,
for $i=1, 2, \ldots, j=1, 2, \ldots\}$

(Notes. Since $B.b_i$'s$\in AI(B)$, $o_{B.b_i}$'s become the same value. We have naturally extended the domain of 'f' from attribute values to attribute values and their positions in an attribute stack.)

(2) If A' occurs in an LR item not in the kernel of PS:
(let (the core of) the LR item be $[B\to\cdot A'\beta]$ (#))

$$\begin{aligned} &F_{PS}(A'.a)\\ &=\{f(e_1, e_2, \ldots)\mid \end{aligned}$$

"$A.a=f(B.b_1, B.b_2, \ldots)$" is the semantic rule for

defining $A'.a$. (Each $B.b_i$ ought to be an inherited attribute of the left side symbol B of the LR item (#).) Let $[C \to \gamma \cdot B'\delta]$ be an LR item that directly derives the above LR item (#) ($B'.b_i \in IN'(PS)$).

$e_i \in F_{PS}(B'.b_i)$, for $i=1, 2, \ldots\}$

Ex 4.1 For the partial state PS of Fig. 2(b),

$$E_{PS}(E.env) = F_{PS}(E^{1,3}.env) \cup F_{PS}(E^{2,1}.env)$$
$$= \{(ASST.env, -2)\}$$

since

$F_{PS}(E^{1,3}.env)$

$= \{(ASST.env, o_{ASST.env}) \mid$ "E.env=ASST.env" is the semantic rule for defining $E^{1,3}.env\}$

$= \{(ASST.env, -2)\}$ by Def 4.2 (1),

and

$F_{PS}(E^{2,1}.env)$

$= \{e_1 \mid$ "$E^{2,1}.env = E^{2,0}.env$" is the semantic rule for defining $E^{2,1}.env$. The LR items $[ASST \to V := \cdot E^{1,3}]$ and $[E^{2,0} \to \cdot E^{2,1} + T^{2,3}]$ directly derive the LR item $[E^{2.0} \to \cdot E^{2,1} + T^{2.3}]$.

Thus, e_1 is in $F_{PS}(E^{1,3}.env)$ or in $F_{PS}(E^{2,1}.env)\}$ by Def 4.2 (2).

$= F_{PS}(E^{1,3}.env) \cup F_{PS}(E^{2,1}.env)$

$= F_{PS}(E^{1,3}.env)$ (smallest solution)

$= \{(ASST.env, -2)\}$ by Def 4.2 (1).

Similarly,

$$E_{PS}(T.env) = F_{PS}(T^{3,1}.env) = F_{PS}(E^{1,3}.env) \cup F_{PS}(E^{2,1}.env)$$
$$= \ldots = \{(ASST.env, -2)\},$$
$$E_{PS}(P.env) = F_{PS}(P^{4,1}.env) \cup F_{PS}(P^{5,1}.env) = F_{PS}(T^{3,1}.env)$$
$$= \ldots = \{(ASST.env, -2)\}.$$

As seen from Ex 4.1, $F_{PS}(A'.a)$ can be computed by traversing in reverse the derivation of LR items in a partial state.

Note 4.1 Assumption 3.1 is necessary for the definition of $E_{PS}(A.a)$ and $F_{PS}(A'.a)$ by Def 4.2, for otherwise $E_{PS}(A.a)$ and $F_{PS}(A'.a)$ may not be represented in terms of input attributes of the partial state.

The following example yields E_{PS} containing more than one expression. This example will be used again later for comparison with the original definition by Jones et al.

Ex 4.2 For an attribute grammar G2 which is

. . .

$A^{1,0} \to X A^{1,2} B^{1,3}$

$\{B^{1,3}.b = A^{1,0}.a\}$ (1)

$A^{2,0} \to A^{2,1} B^{2,2}$

$\{B^{2,2}.b = A^{2,0}.a\}$ (2)

. . .

there is an LR partial state

$$PS = \{A^{1,0} \to X A^{1,2} \cdot B^{1,3},\ A^{2,0} \to A^{2,1} \cdot B^{2,2}, \ldots\} \ (*)$$

$E_{PS}(B.b)$ is

$$E_{PS}(B.b) = F_{PS}(B^{1,3}.b) \cup F_{PS}(B^{2,2}.b)$$
$$= \{(A.a, -2)\} \cup \{(A.a, -1)\}$$
$$= \{(A.a, -2), (A.a, -1)\}$$

Def 4.3 A grammar G is *LR-attributed* iff

(1) G is L-attributed, and

(2) For each LR partial state PS of G, and for each inherited attribute $A.a \in IN(PS)$, $E_{PS}(A.a)$ contains only one expression.

Note 4.2 Clearly the class of LR-attributed grammars is larger than that of S-attributed grammars[10]. As for its relation with L-attributed grammars, see section 6.

Note 4.3 In checking that "$E_{PS}(A.a)$ contains only one expression", the definition of equality becomes a problem. We may rely on mathematical equality. For example, in a grammar

$S \to \alpha A \beta$ $\{A.a = S.s\}$

$A \to B\gamma$ $\{B.b = A.a + 1\}$

$B \to C^{3,1}\delta$ $\{C^{3,1}.c = B.b - 1\}$

$A \to C^{4,1}\eta$ $\{C^{4,1}.c = A.a\}$

. . .

and in a partial state

$PS = \{S \to \alpha \cdot A\beta,\ A \to \cdot B\gamma,\ B \to \cdot C^{3,1}\delta,\ A \to \cdot C^{4,1}\eta, \ldots\}$,

$F_{PS}(C^{3,1}.c) = \ldots = \{((S.s, -k)+1)-1\}$ may be regarded as equal to $F_{PS}(C^{4,1}.c) = \ldots = \{(S.s, -k)\}$, where $k = |\alpha|$. However, in most cases checking by mathematical equality is not necessary in practice.

Ex 4.3 Grammar G1 (Fig. 1) is LR-attributed since

(1) G1 is L-attributed, and

(2) For the partial state PS of Fig. 2(b), $E_{PS}(E.env)$ contains only one expression $\{(ASST.env, -2)\}$ (cf. Ex 4.1). Similar reasoning holds for $E_{PS}(T.env)$, $E_{PS}(P.env)$ and any other partial states.

Ex 4.4 Grammar G2 (Ex 4.2) is not LR-attributed since $E_{PS}(B.b)$ contains two expressions in a partial state (*).

In LR-attributed grammars, the form of each inherited attribute becomes unique in each partial state. Due to this uniqueness, each inherited attribute can be evaluated consistently and stored in a field of the 'ias' stack during LR parsing. (As for synthesized attributes, the uniqueness is clear.) Actually, if $E_{PS_m}(A.a)$ is

$\{f((B.b_1, -k_0), (B.b_2, -k_0), \ldots,$
$(X_1.x_1, -k_1), (X_2.x_2, -k_2), \ldots)\}$

where $B.b_i$'s are inherited attributes and $X_i.x_i$'s are synthesized attributes (the offsets of $B.b_i$'s become the

same by Def 4.2(1)), then the value of $A.a$ in PS_m can be computed by the assignment

ias[top].$A.a$-field:=f(ias[top−k_0].$B.b_1$-field,
ias[top−k_0].$B.b_2$-field, . . .,
sas[top−k_1].$X_1.x_1$-field,
sas[top−k_2].$X_2.x_2$-field, . . .)

Ex 4.5 For G1, assume that we started the analysis from the configuration

(S_0 , . . . a:=b+c; . . . \$)

and reached the configuration

(S_0 . . . $X_m\overline{X}_mS_m\overline{PS}_m$, b+c; . . . \$).

A snapshot of this configuration in the analysis (cf. Fig. 3) is,

S_{m-2}(=ps[top−2])={. . .→. . . X · ASST . . .,
ASST→ · V:=E, . . .}

PS_{m-2} is a subset of S_{m-2} with lookahead 'name'

$\overline{PS}_{m-2}$ (=ias[top−2])=**record** value of ASST.env, etc. **end**

$\overline{X}_{m-2}$ (=sas[top−2])=record containing AS(X)

S_{m-1} (=ps[top−1])={ASST→V · :=E}

PS_{m-1} is the same as S_{m-1} with lookahead ':='

$\overline{PS}_{m-1}$(=ias[top−1])=record containing AI(:=)
=ϕ

$\overline{X}_{m-1}$(=sas[top−1])=record containing AS(V)

S_m(=ps[top]) =Fig. 2(a)
={ASST→V:= ·E, E→ ·E+T,
E→ ·T, T→ ·P**T, T→ ·P,
P→ ·name, P→ ·(E)}

PS_m is a subset of S_m with lookahead 'name'
=Fig. 2(b)
={ASST→V:= ·E, E→ ·E+T,
E→ ·T, T→ ·P**T, T→ ·P,
P→ ·name}

$\overline{PS}_m$ (=ias[top])=**record** value of E.env, value of T.env, and value of P.env **end**

$\overline{X}_m$ (=sas[top])=record containing AS(:=)=ϕ

At the partial state PS_m of Fig. 2(b), E.env, T. env and P.env have the following E_{PS_m}.

E_{PS_m}(E.env)=E_{PS_m}(T.env)=E_{PS_m}(P.env)
={(ASST.env, −2)}

This means that these attribute values can be copied from the 'ASST.env' field of ias[top−2]. That is,

ias[top].E.env-field:=ias[top−2].ASST.env-field;
ias[top].T.env-field:=ias[top−2].ASST.env-field;
ias[top].P.env-field:=ias[top−2].ASST.env-field

Notes 4.4

1. In cases where the semantic rule is a copy rule such as "C.c=B.b", much optimization can be made at attribute evaluation time. For example, let a partial state PS of a grammar G3 be

S→A·B {B.b=f(S.s)}
B→·C {C.c=B.b}
. . .

E_{PS}'s become

E_{PS}(B.b)= {f((S.s, −1))}
E_{PS}(C.c)= {f((S.s, −1))}

by the definition. This usually means the assignments

ias[top].B.b-field:=f(ias[top−1].S.s-field); (*)
ias[top].C.c-field:=f(ias[top−1].S.s-field)

occur at attribute evaluation time. However, optimization into

ias[top].B.b-field:=f(ias[top−1].S.s-field);
ias[top].C.c-field:=ias[top].B.b-field

is easy and more favorable.

2. We assume from the nature of attribute grammars that semantic rules are purely functional, i.e., there are no side effects. But, in cases where there are side effects, some unpredictable problems arise. For example, the assignments (*) for the grammar G3 of the above note 1 show that the function f may be called twice for evaluating B.b and C.c. If f has a side effect, this attribute evaluation may cause multiple side effects even when we want to cause the side effect only once.

3. In Ex 4.5, E_{PS_m}(E.env), E_{PS_m}(T.env) and E_{PS_m}(P.env) have the same value {(ASST.env, −2)}. Optimization by collecting such attributes into an equivalence class will be dealt with in a separate paper [15].

Here, we compare our definition with Jones et al.'s definition [7].

Firstly, the offset in Def 4.2(1) was lacking in the definition of [7], but we have included it since it is necessary in cases such as Ex 4.2: In Ex 4.2, although the semantic rules (1) and (2) are the same, E_{PS}(B.b) shows that there are two possible forms of semantic expressions for B.b since $A^{1,0}$.a of (1) and $A^{2,0}$.a of (2) are in different positions (at 'top−2' and 'top−1', respectively) in the attribute stack at the partial state PS (*). Thus, the value of B.b cannot be computed *uniquely* at this partial state, violating the LR-attributed property. In the original definition of [7], E_{PS}(B.b) was defined in a way that it contains only one expression, E_{PS}(B.b) ={A.a}, thus the grammar is *incorrectly regarded to be LR-attributed.* Ishibashi [3] also remedied this defect by using a homomorphism, but he did not give any concrete form for the homomorphism.

Secondly, we have made the Def 4.2 more precise than Jones et al.'s. We distinguished $E_{PS}(A.a)$ and

$F_{PS}(A'.a)$. We clarified computation by "smallest solution" and consideration of LR item derivation, while their definition was rather vague on these subjects.

Thirdly, Def 4.2 and Def 4.3 are extensions of their definition in that LR partial states are used instead of LR states for attribute evaluation as suggested in their note [7]. By this extension, LR-attributed grammars actually become the maximum class of attribute grammars for which attributes can be evaluated during LR parsing (cf. [17]). For example, the following attribute grammar is LR-attributed if the definition based on LR partial states is used, but it is not LR-attributed if the definition based on LR states is used.

Ex 4.6 Let the grammar G4 be

S→AbB
  {A.ai=1; B.bi=f(. . ., A.as, . . .) . . .}
S→AcB
  {A.ai=2; B.bi=f(. . ., A.as, . . .) . . .}
A→ε
  {A.as=A.ai}
. . .

where A.ai∈AI(A) and A.as∈AS(A).

If we used LR states in the definition of semantic expressions, E_{S_1}(A.ai) contains more than one expression for the LR state S_1={S→·AbB, S→·AcB, A→·}, as

$$E_{S_1}(A.ai)=\{1, 2\}$$

Thus, G4 would not be LR-attributed.

If we use LR partial states in the definition, E_{PS_1}(A.ai) contains only one expression for the partial state PS_1={S→·AbB, A→·} (lookahead=b), which is

$$E_{PS_1}(A.ai)=\{1\}$$

and for the partial state PS_2={S→·AcB, A→·} (lookahead=c), which is

$$E_{PS_2}(A.ai)=\{2\}$$

Thus, G4 is now LR-attributed.

An interesting property which has been recently demonstrated is that any L-attributed LL(1) grammar can always be expressed by an LR-attributed LR(1) grammar [20]. Use of partial states instead of LR states plays an essential role in its proof.

5. Algorithms for Checking the LR-attributed Property and Providing the Semantic Expressions

We give two algorithms for checking whether a given attribute grammar is LR-attributed or not and to provide the semantic expressions $E_{PS}(A.a)$'s. The first is based on the recursive definition of $E_{PS}(A.a)$. The second is non-recursive and works simultaneously with the computation of the closure of LR items.

The first algorithm borrows the concept of PSPG (partial state position graph) from [12]. A PSPG is a directed graph representing a partial state. A node in a PSPG is an item node corresponding to an LR item of the partial state or a special node. A special node is either "I", "SHIFT", or "REDUCE i", which represents an initial node, a shift operation (in parsing), or a reduce operation (in parsing), respectively. There is an edge from an item node to another item node if the former item "directly derives" the latter item in the partial state. There is an edge from "I" to each node corresponding to an item in the kernel. There is an edge from an item node to "SHIFT" if the item is of the form $[A\rightarrow\alpha\cdot a\beta, b]$ where a is a terminal symbol. There is an edge from an item node to "REDUCE i" if the item corresponds to production i and is of the form $[A\rightarrow\alpha\cdot, b]$.

Ex 5.1 The PSPG for the partial state of Fig. 2(b) is shown in Fig. 4.

An algorithm is given by Purdom et al. to investigate positions suitable for calling semantic routines in LR(k) parsers. It can be regarded as a different method for checking the LR-attributed property [3]. It uses a labelling of nodes of PSPG. But their method is not well suited to a practical compiler generator, since it requires rather complicated graph manipulations such as detection of predominators of the final node and checking whether a node "derives itself". Our first algorithm is based on the same PSPG, but is simpler and straightforward. It checks the LR-attributed property by recursively traversing nodes in the PSPG, like a list marking algorithm.

Algorithm 1 (check the LR-attributed property and provide the semantic expressions using PSPG)

Input : Attribute grammar G

Output: Determination of whether the given G is LR-attributed or not, and semantic expressions $E_{PS}(A.a)$'s for each partial state PS and for each $A.a$∈IN(PS).

{main}
{eps[ ] is an array and its domain is IN(PS). eps[$A.a$] ($A.a$∈IN(PS)) represents semantic expression $E_{PS}(A.a)$ which contains only one expression for LR-attributed grammars.}
for each LR state S of G **do**
  for each partial state PS of S **do**
  begin
    make the PSPG;
    for each inherited attribute $A.a$∈IN(PS)
      do eps[$A.a$]:=empty;
    for each LR item k in the kernel of PS **do** trav(k);
    write out eps[$A.a$]'s as semantic expressions $E_{PS}(A.a)$'s.
  end;
procedure trav(i:LR item or special node of PSPG);
begin

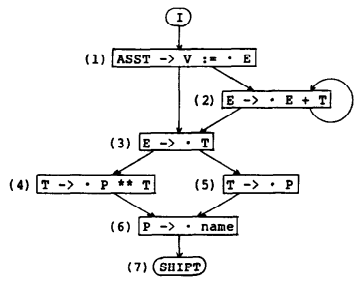


Fig. 4 PSPG of LR partial state of Fig. 2(b).

if (i="REDUCE j") **or** (i="SHIFT") **then** return;
if marked(i) **then** return;
let i **be** of the form $[B\to\alpha\cdot A\beta]$ {only the core[1] is shown here};
{if $i=[B\to\alpha\cdot]$, skip the following for-loop}
for each inherited attribute $A.a$ of A **do**
  begin
    let "$A.a=f(X_1.x_1, X_2.x_2, \ldots)$" **be** the semantic rule defining $A.a$;
    if i is in the kernel **then**
      e:=$f((X_1.x_1, o_{X_1.x_1}), (X_2.x_2, o_{X_2.x_2}), \ldots)$
    else e:=f(eps[$X_1.x_1$], eps[$X_2.x_2$], . . .);
    if eps[$A.a$]=empty **then** eps[$A.a$]:=e
    else if e≠eps[$A.a$] **then**
      "LR-attributed property is violated"
  end;
  mark (i);
  for each LR item ni derived from i **do** trav (ni)
end;

Characteristic features of this algorithm are that the check for the LR-attributed property and the computation of $E_{PS}(A.a)$ (i.e. eps [$A.a$]) are made incrementally by stepwise traversing of the "direct derivation" relation in the closure, starting from the items in the kernel. No repeated traversing of edges of the PSPG is necessary. The time complexity of this algorithm for a partial state PS is linear to the number of inherited attributes in IN′(PS), or roughly linear to the number of nodes of its PSPG.

Notes 5.1

1. In this algorithm, 'eps[$A.a$]' which represents $E_{PS}(A.a)$ corresponds to the smallest solution of $F_{PS}(A'.a)$'s.
2. As for checking the equality of 'e' and 'eps[$A.a$]', see Note 4.3.

Ex 5.2 For the partial state PS of Fig. 2(b), PSPG is already shown in Fig. 4. The algorithm traverses the PSPG, for example, in the order (1), (2), . . ., (7) shown in Fig. 4. At node (1), 'e' becomes (ASST.env, $o_{\text{ASST.env}}$)=(ASST.env, −2) and since eps[E.env] is empty, eps[E.env] becomes (ASST.env, −2). Thus, (ASST.env, −2) is obtained as the semantic expression E_{PS}(E.env). At node (2), 'e' becomes eps[E.env] which is (ASST.env, −2). Since 'e' is equal to eps[E.env], the LR-attributed property is not violated for E.env. It proceeds similarly for node (3) and so on.

Table 1 Results in a description of a Pascal subset.

no. of input lines (excluding type definitions and subordinate routines)	2302
no. of productions	165
no. of nonterminal symbols	64
no. of terminal symbols	66
no. of synthesized attributes	128
no. of inherited attributes	145
(maximum no. of fields in the stack for inherited attributes in LR-attributed grammars)	

Usually, the use of LR states instead of partial states will suffice for attribute evaluation. The second algorithm uses LR states and works simultaneously with the computation of the closure of LR items. This algorithm is non-recursive and does not require the PSPG. A modification of this algorithm was adopted in our compiler generator Rie [4][5][6].

Algorithm 2 (check the LR-attributed property and provide the semantic expressions simultaneously with computation of closure)
Input, Output: same as Algorithm 1 except that $E_S(A.a)$'s are output for each LR *state* S instead of each partial state. ($E_S(A.a)$ is the union of $E_{PS}(A.a)$ for all partial states $PS\subset S$.)

{main}
{es[ ] is an array and its domain is IN(S), where IN(S) is the union of IN(PS) for $PS\subset S$. es[$A.a$] ($A.a\in$IN(S)) represents semantic expression $E_S(A.a)$ which contains only one expression for LR-attributed grammars.}
for each LR state S of G **do**
  begin
    {we assume that the kernel of S is given; the non-kernel of S is derived in this algorithm}
    closure:=empty;
    for each inherited attribute $A.a\in$IN(S) **do**
      es[$A.a$]:=empty;
    for each LR item k in the kernel of S **do**
      begin check(k); closure:=closure ∪{k} **end**;
    repeat
      select LR item i **from** closure;
      let i **be** of the form $[C\to\alpha\cdot B\beta]$ {only the core[1] is shown here};
      {if $i=[C\to\alpha\cdot]$, skip the following for-loop}
      for each production p with B as left side **do**
        begin
          let p **be** of the form "$B\to\gamma$";
          if $[B\to\cdot\gamma]\notin$closure **then**
            begin
              check($[B\to\cdot\gamma]$);
              closure:=closure ∪{$[B\to\cdot\gamma]$}
            end

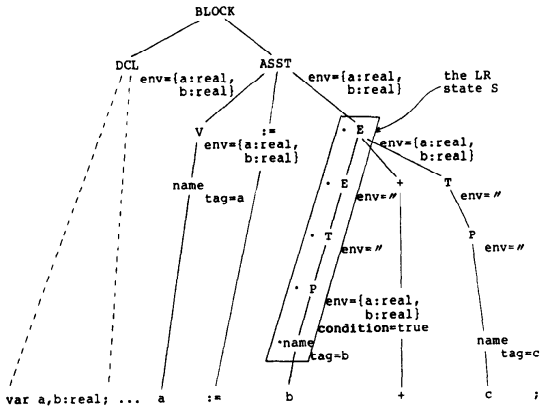


Fig. A1 The concept of LR-attributed grammars.

```
        end
    until all LR items in closure checked;
    write out es[A.a]'s as semantic expressions
    E_S(A.a)'s at S
  end;
procedure check(i: LR item);
begin
  let i be [B→α·Aβ];
  {if i=[B→α·], skip the following for-loop}
  for each inherited attribute A.a of A do
    begin
      let "A.a=f(X1.x1, X2.x2, . . .)" be the semantic
            rule defining A.a;
      if i is in the kernel then
        e:=f((X1.x1, o_{X1.x1}), (X2.x2, o_{X2.x2}), . . .)
      else e:=f(es[X1.x1], es[X2.x2], . . .);
      if es[A.a]=empty then es[A.a]:=e
      else if e≠es[A.a] then
        "LR-attributed property is violated"
    end
end;
```

6. Some Experience

The statistical data in Table 1 was obtained by describing a compiler front-end for a subset of Pascal [2] using our compiler generator. This front-end uses 145 inherited attributes, all of which satisfy the LR-attributed property.

As regards comparison of LR-attributed grammars with other attribute grammar classes, we have no definite results yet. But, writing descriptions of a subset of Pascal and other small languages using LR-attributed grammars was as easy as writing these using L-attributed grammars. Considering that we want to evaluate attributes during left-to-right parsing, the L-attributed property (Def 4.3 (1)) must be inherently satisfied. This property usually holds in most modern programming languages which are designed around the easier one-pass processing technique. The newly imposed restriction (2) of Def 4.3 was not severe, since in many programming languages, inherited attributes such as environment are either simply copied or modified only in block entry to form a nested scope, but both can be easily described so as to satisfy this restriction (2) [2] [16][6].

7. Concluding Remarks

In this paper, we first have provided corrections and improvements for the definition of LR-attributed grammars by Jones and Madsen.

Next, we have presented two practical algorithms for checking the LR-attributed property and providing the semantic expressions of a given attribute grammar. A compiler generator called Rie has been written based on a subclass of LR-attributed grammars, by collecting inherited attributes into equivalence classes. This will be discussed in another paper [15].

An interesting property which has been recently demonstrated is that any L-attributed LL (1) grammar can always be expressed by an LR-attributed LR (1) grammar [20].

As for future problems, it is necessary to examine to what extent the class of LR-attributed grammars can be practically applied, or how strict the restriction of LR-attributed grammars is to compiler writers. The LR-attributed grammar class must also be compared with other classes of attribute grammars and other techniques such as using global attributes to represent some kinds of inherited attributes, e.g. environment.

Acknowledgements

The authors wish to thank Hiroyoshi Ishibashi for introducing us to LR-attributed grammars, Rie Inada for the description of a Pascal subset in our compiler generator system, and David Duncan for polishing up the English in this paper.

This work is supported by the Grant in Aid from the Ministry of Education and Culture, No. 59780016.

Appendix A (The concept of LR-attributed grammars)

Historically, in order that attribute evaluation could be made during bottom-up parsing, a class of attribute grammars called *S-attributed* grammars was first proposed [10]. This class only allows synthesized attributes and excludes any inherited attributes, because it was considered impossible to use inherited attributes since the parsing tree above the analyzed part is not determined. In S-attributed grammars, attributes are evaluated at reduction time where the production to be reduced is determined uniquely.

In 1980 however, Jones and Madsen [7] showed that inherited attributes satisfying certain conditions can also be evaluated during LR parsing, by investigating

the behavior of attributes in LR states. Such inherited attributes are evaluated at the state transition time where the next state is pushed on the parsing stack (not at the reduction time). This class is called *LR-attributed* grammars.

This class has solved the problem in S-attributed grammars, which have to insert nonterminals (with an empty right side) if semantic evaluation should take place in the midst of the right side of productions.

The idea of LR-attributed grammars can be demonstrated as follows using Fig. A1. In Fig. A1, the subtree under ASST is an application of the attribute grammar G1 (Fig. 1). A declaration part is added to provide an explanation. Assume that the LR parsing proceeded until 'b' of 'a:=b+c' is read. At that point, although the subtree under 'E' is not yet determined, the LR state S of Fig. 2(a) can be determined due to the LR parsing method. Thus the inherited attribute ASST.env can be copied down to E.env, T.env and to P.env through this LR state S. That is, information in an ancestor node in the parsing tree can be seen through this LR state. However, a condition must hold in order to enable this. That is, the value of each inherited attribute in this LR state, e.g., E.env, T.env and P.env, must be determined uniquely. This is the informal meaning of Def 4.3.

From the viewpoint of a compiler writer, LR-attributed grammars allow him/her to see all the information from the beginning of the source program up to the point currently being processed. For example, DCL.env can be seen at the present point 'b'. In this sense, LR-attributed grammars are similar to L-attributed grammars where the information at the left of the present point can be utilized [6] [20].

References

1. AHO, A. V., SETHI, R. and ULLMAN, J. D., Compilers-Principles, Techniques, and Tools, (Addison-Wesley) (1985).
2. INADA, R., Design of a Semantic Description Language Based on Attribute Grammars and its Application to Pascal, Bachelor Thesis, *College of Inf. Sci., Univ. of Tsukuba* (1984) (in Japanese).
3. ISHIBASHI, H., On Syntax Analyses in View of Semantic Analysis, Master Thesis, *Dept. of Inf. Sci., Tokyo Institute of Technology* (1982) (in Japanese).
4. ISHIZUKA, H., A Compiler Generator Based on an Attribute Grammar Suitable for LR Parsing, Bachelor Thesis, *College of Inf. Sci., Univ. of Tsukuba* (1984) (in Japanese).
5. ISHIZUKA, H. and SASSA, M., A Compiler Generator Based on "ecLR-attributed" Grammars, *Proc.* 29th *Ann. Conv. IPSJ* (*Inf. Proc. Soc. Japan*), 4D-12 (Sep. 1984).
6. ISHIZUKA, H. and SASSA, M., A Compiler Generator Based on an Attribute Grammar, *Proc.* 26th *Programming Symposium of IPSJ*, Hakone, (Jan. 1985) 69-80 (in Japanese).
7. JONES, N. D. and Madsen, M., Attribute-influenced LR Parsing, *Lecture Notes in Comp. Sci.* 94 (Springer) (1980) 393-407.
8. KASTENS, U., HUTT, B. and ZIMMERMANN, E., GAG: A Practical Compiler Generator, *Lecture Notes in Comp. Sci.*, 141 (Springer) (1982).
9. KNUTH, D. E., Semantics of context-free languages, *Math. Syst. Theory*, **2**, 2, (1968) 127-145 and (correction) *ibid*, **5**, 1, (1971) 95-96.
10. LEWIS II, P., Rosenkrantz, D. and Stearns, R., Compiler Design Theory (Addison-Wesley) (1976).
11. POHLMANN, W., LR Parsing for Affix Grammars, *Acta Inf.*, **20** (1983) 283-300.
12. PURDOM, P. and BROWN, C. A., Semantic Routines and LR(k) Parsers, *Acta Inf.*, **14**, (1980) 299-315.
13. RÄIHÄ, K-J. el al., The Compiler Writing System HLP, Report A-1978-2, *Dept. of Comp. Sci., Univ. of Helsinki*, Finland (1978).
14. SASSA, M., Compiler Generators, *J. IPS Japan* (Johoshori), **23**, 9, (Sep. 1982) 802-817 (in Japanese).
15. SASSA, M., ISHIZUKA, H. and NAKATA, I., ECLR-attributed Grammars: A Practical Class of LR-attributed Grammars, to appear in *Inf. Process. Lett.* (1986).
16. SASSA, M. and ISHIZUKA, H., Description of PL/0 Compiler by ecLR-attributed Grammars, Tech. Memo PL-2, *Inst. of Inf. Sci., Univ. of Tsukuba* (1984) (in Japanese).
17. TARHIO, J., Attribute Evaluation during LR Parsing, Rep. A-1982-4, *Dept. of Comp. Sci., Univ. of Helsinki*, Finland (1982).
18. WAITE, W. M. and GOOS, G., Compiler Construction, Chap. 8 (Springer) (1984).
19. WATT, D. A., The Parsing Problem for Affix Grammars, *Acta Inf.*, **8**, (1977) 1-20.
20. NAKATA, I. and SASSA, M., L-attributed LL (1) Grammars are LR-attributed, to appear in *Inf. Process. Lett.* (1986).

(Received December 24, 1984; revised September 3, 1985)