ISE

ISE-TR-90-82

Rie — Introduction and User's Manual

by

Masataka Sassa, Harushi Ishizuka,
Motoki Sawatani, and Ikuo Nakata

June 20, 1990

INSTITUTE
OF
INFORMATION SCIENCES AND ELECTRONICS
UNIVERSITY OF TSUKUBA

Rie - Introduction and User's Manual

Masataka Sassa, Harushi Ishizuka,
Motoki Sawatani and Ikuo Nakata

Tech. Report ISE-TR-90-82 (June 1990)

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

This manual is an extended and revised version of "Rie user's manual (in Japanese)", Tech. Memo PL-4, Univ. of Tsukuba (1985).

Table of Contents

1 Introduction

This manual introduces Rie [Ishizuka et al. 85, Sassa et al. 85a], a compiler generator based on attribute grammars [Knuth 68].

Attribute grammars have been used as the basis of many compiler generators: GAG [Kastens et al. 82], Linguist-86 [Farrow 84], HLP84 [Koskimies et al. 86] or Synthesizer Generator [Reps 89]. Compared to them, Rie is made according to the following observations.

The class of attribute grammars for which attributes can be evaluated in a single pass during parsing without making a parse tree is called one-pass attribute grammars. This class is worth attention since most modern programming languages are now designed using the one-pass processing techniques. Among one-pass attribute grammars, **LR-attributed grammar** [Jones et al. 1980, Sassa et al. 85b], which is the class of attribute grammars for which attributes can be evaluated during LR parsing, can be regarded as one of the largest classes concerning translation power [Nakata et al. 86].

Rie is based on a variant of LR-attributed grammar where optimization of attribute evaluation is achieved by introducing a kind of equivalence relation among inherited attributes. This class is called **ECLR-attributed grammar** [Sassa et al. 87], where EC stands for equivalence class.

From user's viewpoint, Rie can be regarded as an extension of Yacc [Johnson 75], a compiler generator running on the Unix operating system. The description format is made similar to that of Yacc. So, it is easy for readers familiar with Yacc to write Rie descriptions. Rie also inherits implementation techniques from Yacc. The syntax analysis part of the generated compiler is based on LALR(1) grammar with disambiguating rules. The base language of the semantic part is the programming language C, and types, data structures and functions supported by C can be used in semantic rules.

Rie is currently running on many machines, mainly on Unix operating systems - VAX11/750, Micro VAX, SUN-2, 3 and 4, Ustation (a Japanese machine) and on Mac II too. Collaborating with the Unix tools such as Lex is quite easy. Since the system is completely written in C, it can be easily transported to other machines.

Rie has been applied to write a compiler for PL/0 [Sassa et al. 84], a Pascal semantic analyzer which checks semantics based on ISO standard [BSI 82], and a translator for a stream-based language called Stella (= Pascal + stream) into Pascal.

In the following,we first introduce attribute grammars and ECLR - attributed grammars in section 2, and then we present description format and operating instruction of Rie in section 3.

2 Outline of attribute grammars and ECLR-attributed grammars

In this section, we introduce attribute grammars and the LR- and ECLR-attributed grammars which form the basis for the Rie system.

2.1 Attribute grammars

A programming language is usually defined by its syntax and semantics.

Syntax of programming languages can be described using context free grammars. The (static) semantics of a programming language means for example, that only declared variables can be used, or that the exact meaning of operations like + or * is defined according to types of their operands.

Attribute grammar [Knuth 68, Sassa 86, Katayama 87] is one of typical formalisms to express semantics, and is fairly suitable for generation of compilers from a specification. Its description unites both syntax and semantics. Let us explain the concept of attribute grammar using an example.

The example we use here is a simple programing language which involves only declaration and use of variables. For simplicity, there is no nested scope rules for variables. A variable is declared by "dcl id ;" and it is used by "use id ;", where id means a variable. A program is made from declaration of variables followed by statement part (use of variables). The semantics of this language requires that (i) the same variable can not be declared twice, and (ii) only declared variables can be used. If the source program violates these rules, error messages will be printed out. For example,

```
dcl x ; dcl y ; dcl x ; use x ;                    (S1)
                ↑error: redeclaration
dcl x ; dcl y ; use y ; use z ;                    (S2)
                        ↑error: use of undeclared variable
```

The syntax of such a language is defined by a context free grammar as follows:

grammar G1:

program	→	dclist stlist	(1)
dclist	→	dclist dc	(2)
dclist	→	dc	(3)
dc	→	dcl id ;	(4)
stlist	→	stlist st	(5)
stlist	→	st	(6)
st	→	use id ;	(7)

In this grammar, 'program', 'dclist', 'dc', 'stlist' and 'st' are nonterminals. 'dclist' represents the declaration part and 'stlist' represents the statement part. 'dcl', 'use', 'id' and ';' are terminals.

Source programs (S1) and (S2) are syntactically correct according to the grammar G1, but they contain semantic errors. In order to check this, we can make semantic analysis by using information such as names of variables and set of declared variables. This information is formalized as **attributes** in attribute grammars. For example, we can use 'name' as an attribute representing the name of a variable, and 'env' (shorthand for environment) as the set of declared variables or the symbol table.

We can think conceptually that a parse tree is already made by performing syntax analysis. Attributes can be associated with each node of the parse tree. The values of attributes are computed in that parse tree. (Note. Actually the parse tree may not be created in the case of one-pass attribute grammars.)

The attributed parse tree for the source program (S2) is shown in Fig. 1. Normally, we assume that values of attributes attached to terminals are available as a result of lexical analysis. For example here we assume that values of attributes 'name' attached to terminal 'id' are obtained from the lexical analyzer.

Computation of attribute values is called **attribute evaluation**, and a procedure which performs attribute evaluation is called **attribute evaluator**. Let us look how attriute evaluation takes place in Fig. 1. Roughly speaking, the attribute evaluator first

collects information about declared variables below 'dclist' (which is a son of 'program'), and then transfer this information to 'stlist'.

More detailed, values of attributes in this example are evaluated from (1) to (8). First at (1), 'name' becomes 'x'. Then at (2), 'env' becomes {x}, which means that the symbol table now contains 'x'. At (3), 'name' = y. At (4), 'env' = {x,y} by taking the union of env of (2) = {x} and name of (3) = y. Attributes whose values can be computed from sons to father on the parse tree like 'name' of 'dc' and 'env' of 'dclist' are called **synthesized attributes.**

Next, we let the value of 'env' of 'stlist' at (5) be the same as the value of 'env' of 'dclist' at (4) = {x,y}. Similarly, we let 'env' at (6) and (7) be copies of the same value. Attributes whose values can be computed from father or from brothers like 'env' of 'stlist' or 'env' of 'st' are called **inherited attributes.**

Now at (iii) variable 'y' is used. Since 'name' at (iii) = y is included in 'env' at (7) = {x,y}, this is correct. On the other hand, variable 'z' is used at (iv). This turns out to be an error of using undeclared variable since 'name' at (iv) = z is not included in 'env' at (8) = {x,y}.

We can express the above semantic analysis formally as an attribute grammar as follows.

attribute grammar AG1:

program	→	dclist stlist	(1)
		{ stlist.env = dclist.env }	(1a)
dclist	→	dclist dc	(2)
		{ $dclist_1$.env = $dclist_2$.env ∪ {dc.name} ;	(2a)
		condition dc.name ∉ $dclist_2$.env	(2b)
		message redeclaration of identifier }	
dclist	→	dc	(3)
		{ dclist.env = {dc.name} }	(3a)
dc	→	dcl id ;	(4)
		{ dc.name = id.name }	(4a)
stlist	→	stlist st	(5)
		{ $stlist_2$.env = $stlist_1$.env ;	(5a)
		st.env = $stlist_1$.env }	(5b)
stlist	→	st	(6)
		{ st.env = stl ist.env }	(6a)
st	→	use id ;	(7)
		{ **condition** id.name ∈ st.env ;	(7a)
		message identifier not declared }	

An attribute grammar unifies both syntax and semantics. Semantic rules are represented here by enclosing them by { and }. Attributes can be associated with grammar symbols. Semantic rules indicate how attributes of nonterminals in a production can be computed from attributes of other grammar symbols in the same production. Notation like 'id.name' means attribute 'name' associated with 'id'.

Semantic rule (1a) is associated with production (1) and represents that the symbol table 'dclist.env' collected in the declaration part is copied to 'stlist.env' as the symbol table of the statement part. In semantic rule (2a), '$dclist_1$' and '$dclist_2$' denote the first and second 'dclist' in the corresponding production rule (2), respectively. We use subscripts like $_{1,2}$ to discriminate occurrences of a grammar symbol in a production. Semantic rule (2a) means that the union of the symbol table of the previous declaration part '$dclist_2$.env' and the name of the newly declared variable 'dc.name' becomes the symbol table of the new declaration part '$dclist_1$.env'. That is, declared variables are collected into the symbol table.

The '**condition** ... **message** ...' in (2b) is called a **context condition**, and declares semantical condition to be satisfied and an error message for the violation. For example, context condition (2b) represents that, 'dc.name', the name of the newly declared

variable, must be different from '$dclist_2$.env', the set of variable names already declared. (Note that the notion of context condition differs from context-dependent parsing. The context condition does not affect parsing. We do not deal with context-dependent or attribute-influenced parsing here, although it is also an interesting theme.)

An important characteristic of attribute grammar is that attribute evaluation is made independent of the order of description of semantic rules. Rather, starting from the point where attribute values are defined, attribute evaluation proceeds according to the dependency among attributes in semantic rules. That is, the value of the attribute in the left hand side of a semantic rule is evaluated after all values of attributes in the right hand side are obtained. In other words, values of attributes are evaluated only once according to the corresponding semantic rules, and the values are invariable after evaluation. In this sense, we can say that attribute grammars are **functional**, or have the property of **single assignment**. They do not allow side-effects. (In practical grammars however, side-effects might be useful).

Now, we can define attribute grammars more formally.

Def. An attribute grammar G is defined by the following (1) - (3).
(1) Context free grammar $G_u = (N, T, P, Z)$ which is called 'underlying context-free grammar'. This represents the syntactic part of the attribute grammar. Here, N is a set of nonterminals, T a set of terminals, P a set of productions, and Z the start symbol. We let $V = N \cup T$.
(2) To each grammar symbol $X \in V$, are associated two disjoint finite sets S(X) and I(X). S(X) and I(X) are the set of synthesized attributes and inherited attributes, respectively. That is, an attribute is either synthesized or inherited.
An attribute a of symbol X can be represented as X.a.
(3) For each production $p \in P$ there is a set of semantic rules. Let p be

$$p : X_0 \rightarrow X_1 \ldots X_{np}$$

and a semantic rule associated with p be

$$X_k.a = f(X_{i1}.a_1 , \ldots , X_{im}.a_m)$$

where $1 \leq k, i_1, \ldots, i_m \leq np$. Then,

$$X_k.a \in S(X_0) \cup I(X_1) \cup \ldots \cup I(X_{np})$$

That is, the left hand side of a semantic rule must be a synthesized attribute of the 'father' or an inherited attribute of a 'son'. In other words, whether an attribute is synthesized or inherited can also be determined from semantic rules.
$X_k.a$ or $X_{ij}.a_j$ is called an **attribute occurrence** of the production.
In the semantic rule above, we say that $X_k.a$ **depends** (or, has a **dependency**) on $X_{i1}.a_1, \ldots, X_{im}.a_m$ in p.
(We omit the definition about context condition. But its meaning should be clear.)

Notes:
1. If grammar symbols are different, attributes associated with them are regarded to be different, although their values might be the same. For example, 'dclist.env' and 'stlist.env' are different attributes.
2. In this report, we use the term 'attribute' informally. It often means 'attribute occurrence' or 'attribute instance' in the formal terminology.

We also define Bochmann normal form as follows.

Def. When each attribute $X_{ij}.a_j$ appearing in the right hand side of a semantic rule satisfies

$$X_{ij}.a_j \in I(X_0) \cup S(X_1) \cup \ldots \cup S(X_{np})$$

and if this holds for every semantic rules,then the grammar is said to be in **Bochmann normal form**.

2.2 LR-attributed grammars and ECLR-attributed grammars

2.2.1 LR-attributed grammars

Here, we outline how attribute evaluation can take place during LR parsing. Further details can be found in [Sassa et al. 85b]. Henceforth we assume that grammars are in **Bochmann normal form.**

First, since attributes must be evaluated from left to right, dependencies among attributes in semantic rules should also be from left to right. This is known as the L-attributed property.

Def. Attribute grammar G is called **L-attributed**, iff for any production $X_0 \rightarrow X_1 \ldots X_{np}$ the following condition holds.
Each attribute in $I(X_k)$ ($1 \leq k \leq np$) depends only on attributes in

$$I(X_0) \cup S(X_1) \cup \ldots \cup S(X_{k-1}).$$

This definition means that for defining the value of an inherited attribute of a son X_k in the parse tree, we can only use inherited attributes of its father X_0 or synthesized attributes of its left brothers $X_1, \ldots, X_{k-1}$. No restriction is imposed for defining the value of a synthesized attribute of the father X_0.

As an example, attribute grammar AG1 is L-attributed.
But, if (1) and (1a) of AG1 were

program → dclist stlist (1)
{ dclist.a = stlist.a } (1a')

the grammar would no longer be L-attributed since there is a right to left dependency.

The L-attributed property is usually known to be the condition to evaluate attributes during LL parsing.

Secondly, let us see how attributes, especially inherited attributes, can be evaluated during LR parsing. The reader may wonder whether it is possible to evaluate inherited attributes, which are attributes whose values are determined from top to bottom, during LR parsing, which is a bottom-up parsing. But we can show that if inherited attributes satisfy a certain condition, we can truly evaluate their values during LR parsing. The key is in the concept of LR states. (An LR state is the canonical collection of LR items [Aho et al. 86].)

Assume that the LR parsing has proceeded until position (*) just before reading 'use y' in source program

dcl x ; dcl y ; use y ; use z ; (S2)
↑ (*)

The status of the parser can be thought to be box (a) in Fig. 2. But, since at this moment (*), we have not yet reduced 'use id ;' to 'st', we do not know what nodes will come above 'use id ;' and what will be the concrete form of the parse tree in part (b) of the figure. On the other hand the parser only knows that we are in the following LR state.

LR state I_1:

[$program^{1,1}$ → $dclist^{1,2}$. $stlist^{1,3}$] (K1)
[$stlist^{2,1}$ → . $stlist^{2,2}$ $st^{2,3}$]
[$stlist^{3,1}$ → . $st^{3,2}$]
[$st^{4,1}$ → . $use^{4,2}$ $id^{4,3}$;]

(We omit the lookahead part of LR items for simplicity. Superscripts are added for a later explanation.)

An LR state is made in such a way that LR items in it represent all possibilities at a certain parsing situation. Thus, although we do not know the actual form of the parse tree, we can know what possible forms the parse tree will have when analysis proceeds further. In other words, an LR state contains information necessary to predict possible forms of the forthcoming parse tree. Therefore, if values of inherited attributes can be determined

uniquely and consistently in the present LR state, we are really able to evaluate inherited attribute values.

We give some terminology. The set of LR items in an LR state which have their '.' (dot or LR marker) not at the beginning of the right hand side is called **kernel** of the LR state. For example LR item (K1) of I_1 above is in the kernel.

Now, let us see how to evaluate inherited attributes. Let us take 'stlist.env'. In Fig. 2, it would be either 'stlist^1.env' or 'stlist^2.env' or 'stlist^3.env'. If we knew the exact parse tree of Fig. 2, we could identify one of them, for example 'stlist^3.env', and its value would be evaluated as follows:

$$\text{stlist}^3.\text{env} = \text{stlist}^2.\text{env} = \text{stlist}^1.\text{env} = \text{dclist}^1.\text{env}$$

However, since we do not know the exact parse tree at this moment, we can not distinguish among 'stlist^1', 'stlist^2' and 'stlist^3'. This is because 'stlist^2' and 'stlist^3' might not appear or there might be more 'stlist's like 'stlist^4', 'stlist^5' etc. in the actual parse tree. From the way of constructing LR states, we can only say that 'stlist^1' of Fig. 2 corresponds to 'stlist^{1,3}' in I_1, and 'stlist^2' and 'stlist^3' correspond to 'stlist^{2,2}' in I_1. Thus, we try to evaluate both attributes 'stlist^{1,3}.env' and 'stlist^{2,2}.env' in the present LR state I_1. By backtracing the derivation of LR items in an LR state, we get:

$$\text{stlist}^{1,3}.\text{env} = \text{dclist}^{1,2}.\text{env} = \{x,y\}$$

and

$$\text{stlist}^{2,2}.\text{env} = \text{stlist}^{2,1}.\text{env}$$
$$(= \text{stlist}^{2,2}.\text{env} = \text{stlist}^{2,1}.\text{env} = \ldots)$$
$$= \text{stlist}^{1,3}.\text{env} = \text{dclist}^{1,2}.\text{env} = \{x,y\}$$

Fortunately, we got the same value for both 'stlist^{1,3}.env' and 'stlist^{2,2}.env'. Thus, we can see that the value of 'stlist.env' is

$$\text{stlist.env} = \text{dclist}^{1,2}.\text{env} = \{x,y\}$$

From another viewpoint, this can be said that we have succeeded in evaluating an inherited attribute using information from an ancestor node in the parse tree, although we do not know the exact form of the parse tree.

Next, let us take 'st.env'. This corresponds to 'st^{3,2}.env' in I_1. Its value is evaluated similarly as before:

$$\text{st}^{3,2}.\text{env} = \text{stlist}^{3,1}.\text{env}$$
$$(= \text{stlist}^{2,2}.\text{env} = \text{stlit}^{2,1}.\text{env} = \ldots.)$$
$$= \text{stlist}^{1,3}.\text{env} = \text{dclist}^{1,2}.\text{env} = \{x,y\}$$

Note that getting the value of 'st^{3,2}.env' has been possible since we got the same value for both 'stlist^{2,2}.env' and 'stlist^{1,3}.env'. Thus,

$$\text{st.env} = \text{dclist}^{1,2}.\text{env} = \{x,y\}$$

This completes the outline of evaluation of inherited attributes during LR parsing.

Let us now see a little more in detail how we can check the condition that inherited attributes can be evaluated uniquely without inconsistency.

First, we define the set **IN**. This represents the set of inherited attributes to be evaluated at a given LR state. This is a set of inherited attributes of nonterminals after the '.' (dot or LR marker) in the LR state. That is , if I_i is an LR state,

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

For example, $IN(I_1)$ of the above LR state I_1 is { stlist.env, st.env }.

Next, in order to describe that attributes can be evaluated 'uniquely', recall that we got

$$\text{stlist.env} = \text{dclist}^{1,2}.\text{env}$$

or

$$\text{st.env} = \text{dclist}^{1,2}.\text{env}$$

in the above example. Noting that 'dclist^{1,2}' is an LR item in the kernel, 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 represented by the 'semantic expression'. The **semantic expression** $E_{Ii}(A.a)$ of an inherited attribute A.a in $IN(I_i)$ of LR state I_i is a <u>set</u> of possible expressions or symbolical forms of functions for evaluating A.a in terms of atrributes of LR item(s) in the kernel of I_i. For example,

$$E_{I1}(\text{stlist.env}) = \{ \text{expr. for evaluating 'stlist}^{1,3}\text{.env'} \} \cup \{ \text{expr. for evaluating 'stlist}^{2,2}\text{.env'} \} = \{ \text{dclist}^{1,2}\text{.env} \}$$

$$E_{I1}(\text{st.env}) = \{ \text{expr. for evaluating 'st}^{3,2}\text{.env'} \} = \{ \text{dclist}^{1,2}\text{.env} \}$$

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

Now, we can define **LR-attributed grammar** (LR-AG) as follows.

Def. A grammar G is LR-attributed iff

(1) G is L-attributed, and

(2) for each LR state I_i of G, and for each inherited attribute $A.a \in IN(I_i)$, semantic expression $E_{Ii}(A.a)$ contains only one expression (i.e. it is unique).

As an example, grammar AG1 is LR-AG, since

(1) AG1 is L-attributed, and

(2) for LR state I_1, E_{I1}(stlist.env) contains only one expression {dclist^{1,2}.env}. Similar reasoning holds for E_{I1} (st.env) and for any other LR states.

As a counter example which is not LR-attributed, let us change one of semantic rules of AG1 as follows (although this example is quite nonsense)

$$\text{stlist} \rightarrow \text{stlist st} \qquad (5)$$
$$\{ \text{stlist}^2\text{.env} = \text{stlist}^1\text{.env} \cup \{ \text{nil} \} ; \qquad (5a')$$
$$\text{st.env} = \text{stlist}^1\text{.env} \} \qquad (5b)$$

The grammar is no more LR-AG since there is a semantic expression which is not unique as follows:

$$E_{I1}(\text{stlist.env}) = \{ \text{expr. for evaluating 'stlist}^{1,3}\text{.env'} \} \cup \{ \text{expr. for evaluating 'stlist}^{2,2}\text{.env'} \} \text{ (same as before)} = \{ \text{dc}^{1,2}\text{.env} , \text{dc}^{1,2}\text{.env} \cup \{\text{nil}\} \}$$

This completes the definition of LR-attributed grammars.

Now, let us see what the implementation looks like. The configuration for the parser and the attribute evaluator for LR-AG is sketched in Fig. 3. In addition to the usual parsing stack whose elements contain LR states, we use two attribute stacks which behave synchronously with the parsing stack. One is for synthesized attributes, and the other is for inherited attributes. The inherited attribute stack has a field for each inherited attribute. For grammar AG1, for example, we will use a stack like

```
inh_attr_stack : array [1 .. n] of
        record  stlist_env: envtype ;  st_env: envtype  end ;
```

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

(i) Synthesized attributes of a nonterminal A are evaluated when the parser is at an LR state containing an LR item [A → α .] for some α and the parser reduces α to A by production 'A → α'.

(ii) Inherited attributes of a nonterminal B are evaluated when the parser goes to an LR state which contains an LR item [A → β . B γ] with some A, β and γ.

The latter means that if the parser goes to a new LR state I_m (of Fig. 3), $IN(I_m)$ are evaluated and stored into the inherited attribute stack. Thus, attribute evaluation takes place not only at reduction time, but <u>also at state transition time</u>. For example the evaluation of attributes 'stlist.env' and 'st.env' occurs when the parser makes a state transition by nonterminal 'dclist' and go to LR state I_1. This means that we can make 'semantic action' (in traditional terminology) in the midst of the right hand side of a

production. This is a nice feature, since traditionally one had to insert the so-called 'marker nonterminals' (with an empty right-hand side) in such cases only just to perform semantic action.

We have omitted some details like 'offset' and 'LR partial state' in the above explanation. Interested readers can refer to [Sassa et al. 85b].

2.2.2 ECLR-attributed grammars

In LR-AGs, each inherited attribute is assumed to be evaluated and stored separately. But it is often so that many inherited attributes related to the symbol table have the same value except when a new scope or block begins. For example in AG1, 'stlist.env' and 'st.env' have always the same value since the semantic rules between them are copy rules.

Considering this fact, we can easily perform optimization in attribute evaluation. The idea is to collect the set of inherited attributes which have always the same value into certain equivalence classes, and make attribute evaluation in the unit of equivalence classes. Thus, the values of inherited attributes in a same equivalence class can be obtained by a single evaluation and can be stored in a single location at each evaluation point.

Let $EC = \{ EC_1, EC_2, \ldots, EC_n \}$ be a disjoint partition of the set of all inherited attributes of a given grammar. Each EC_i is called an **equivalence class**. It is 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 = {stlist.env, st.env} for grammar AG1. By treating attributes in each equivalence class together, we get a variant of LR-AG called **equivalence class LR-attributed grammar** or **ECLR-attributed grammar** (ECLR-AG). The definition of ECLR-AG is got by substituting equivalence classes to inherited attributes in the definition of LR-AG.

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 $E_{Ij}(A.a)$'s are the same and unique (i.e. contain only one expression) for all inherited attributes $A.a \in EC_i \cap IN(I_j)$.

For the moment, we assume that the partition is given by the user.

As an example, grammar AG1 is ECLR-AG with respect to $EC = \{EC_1\}$, EC_1= {stlist.env, st.env}, since
(1) AG1 is L-attributed, and
(2) for LR state I_1, $E_{I1}(A.a)$'s are $\{dclist^{1,2}.env\}$ and are the same and unique for all inherited attributes $A.a \in EC_1 \cap IN(I_1)$ = {stlist.env, st.env}. Similar reasoning holds for other LR states.

From the definition, we can easily show that the class of ECLR-AGs is not smaller than that of LR-AGs.

The implementation of the attribute evaluator for ECLR-AG is similar to that for LR-AG. As for the inherited attribute stack, instead of allocating a field to each inherited attribute, we allocate a field to each equivalence class. For grammar AG1, for example, we will use a stack like

```
inh_attr_stack : array[1 .. n] of
                 record equiv_class_1: envtype end ;
```

An alternative way of implementation is to use an inherited attribute stack for each equivalence class as shown in Fig.4. Actually, this was adopted in the implementation of Rie. In this figure, $E_{i,m}$ contains the values of inherited attributes in $EC_i \cap IN(I_m)$ whose value must be unique if the grammar is ECLR-attributed.

We have had some results about the optimization effect of collecting inherited attributes into equivalence classes. Space reduction from 1/17 to 1/9 was achieved for inherited attribute stacks in a compiler based on ECLR-AGs compared to the one based on LR-AGs. As for the time of attribute evaluation, a reduction of about 8 percent was achieved compared to LR-AG.

Further details on ECLR-AGs can be found in [Sassa et al. 87].

A system for automatically partitioning inherited attributes into equivalence classes has been implemented as an application of the coloring problem in the graph theory [Yamashita et al. 87].

Finally, what kind is 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. 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 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 condition (2).

3. Rie Manual

3.1 Outline of the Rie system

The Rie system is organized as shown in Fig. 5. The following summarizes each part.

Lexical analysis:
The Rie system does not support lexical analysis. But, any commonly available lexical analyzer generator can be used, such as Lex [Lesk 75] in the Unix operating system. Connecton to Lex is easy as in the case of Yacc, and the interface is described later.

Attribute grammar:
Syntactic and semantic specification of the language is to be described by an ECLR-AG introduced in the previous section. The attribute grammar is generally assumed to be in Bochmann normal form. Extended features, such as local attributes and short-hand notation are provided. In semantic rules, facilities of the C language can be used. The actual format of specification will be given later.

Syntactic and semantic analyzer generator:
The Rie generator can be divided into two parts, syntactic analyzer generator and semantic analyzer generator. The syntactic analyzer generator part inherits many external features from Yacc [Johnson 75]. Semantic analyzer generator part generates attribute evaluator of ECLR-AG described in the previous section.

Syntactic analysis (specification, generator and analyzer):
Let us have a look at the part concerning syntactic analysis. Since the external feature of this part is based on Yacc, the reader are referred to [Johnson 75] for further details.

The specification for syntactic analysis has to be in LALR (1) grammar. When the specification contains an error violating the format of specification, Yacc stops processing immediately. But Rie improves this and it continues processing in many cases.

The syntactic part supports the following 'disambiguating' features which are also inherited from Yacc:
- precedence of operators and production rules
- associativity of operators

They can be utilized to deal with ambiguity of the grammar, which is normally revealed by the generator as shift/reduce conflicts or reduce/reduce conflicts. The ambiguity can often be resolved using the above features, without modifying the grammar itself. Sometimes disambiguating rules are used positively to simplify writing a grammar, such as in syntactic rules for expressions.

The syntactic part also supports the 'error' token, as in Yacc. An 'error' token is a special terminal which can be embedded in a production rule, in order to explicitly specifying error recovery in case of syntactical errors.

The syntax analyzer generated is in the C programming language.

Semantic analyzer generator and attribute evaluator:
Attribute evaluation is made in parallel with LR parsing without making the parse tree. Evaluation takes place at the moment of LR state transitions or reductions.

The attribute evaluator generated is in the C programming language.

Rie also provides runtime options, which are similar to Yacc. They are also described later.

Note. Major difference with Yacc:

In Yacc, semantic analysis is described by action routines attached to each production. They are program fragments made of procedural statements, and are executed when a reduction by the relevant production occurs. Rie essentially differs from Yacc in that semantic analysis is described formally and functionally by an attribute grammar, and that attribute evaluation also takes place in conjunction with LR state transition.

3.2 Specification format

3.2.1 Overall structure

Specification format of Rie inherits from Yacc and ALADIN [Kastens et al. 82]. An example specification will be shown later in section 3.4(2). A specification is divided into four major parts as follows, which are separated by markers `%{`, `%}` or `%%`.

```
%{
    C fragment for the definition of types for attributes
%}
    declaration of grammar symbols, attributes, equivalence classes etc.
%%
    syntactic and semantic rules
%%
    C fragment for user-defined functions
```

3.2.2 C fragment for the definition of types for attributes

In the declaration of attributes, which will be described in the next section, the type of an attribute must be specified by 'type_name'. So, *type_names* used in the declaration of attributes must be defined here beforehand. Types defined here can be any types allowed in the C language. The definitions of types are made using the '`typedef`' clause of the C language. In the example of section 3.4(2) (shown later)

```
typedef  int    ENVTYPE ;
typedef  char   *NAMETYPE ;
```

define `ENVTYPE` as `int` and `NAMETYPE` as a pointer to `char` type.

This part, which is enclosed by `%{` and `%}`, becomes a part of the generated compiler (written in C) without any change. So, any facility supported in C can be described as well, like

```
#define  ENVTABLESIZE  100
```

or, variable or data structure declaration, like

```
NAMETYPE  envtable[ENVTABLESIZE] ;
```

We can also put type definitions in a separate file (for example in a file named `typedef.h`) and just write

```
#include "typedef.h"
```

here.

3.2.3 Declaration of grammar symbols, attributes, equivalence classes etc.

From now on, we use the following notation to describe the specification format.

'A'	A itself, i.e. A is a terminal in the meta language
A	A represents some set of symbols, i.e. A is a nonterminal in the metalanguage
ε	empty string
$(\alpha \mid \beta)$	α or β where α, β are sentential forms of the metalanguage
$[\alpha]$	α or ε , that is $(\alpha \mid \varepsilon)$
$\{\alpha \parallel \beta\}$	It is usually a short-hand for a sequence of α separated by β (including empty string), that is, $\varepsilon, \alpha, \alpha\beta\alpha, \alpha\beta\alpha\beta\alpha, \ldots$
$\{\alpha \parallel \beta\}+$	Similar to the above without empty string, that is, $\alpha, \alpha\beta\alpha, \alpha\beta\alpha\beta\alpha, \ldots$
$A : \alpha$	A represents α, where α is a sentential form of the metalanguage

3.2.3.1 Declaration of grammar symbols and attributes

In this part, we declare grammar symbols, i.e. terminals and nonterminals, and their attributes. The specification format is as follows.

```
( '%terminal' { terminal_name  || ',' }+
| '%nonterm' { nontermial_name  || '.' }+
)
( ':' { attribute_name ':' type_name ( 'inh' | 'synt' | ε ) || ',' }+
| ε
)
';'
```

For instance, in the example of section 3.4(2),

```
%nonterm  stlist, st :
                  env : ENVTYPE  inh ;
```

means that nonterminals `stlist` and `st` have an attribute `env` of type `ENVTYPE` which is an inherited attribute. Types of attributes must be either a basic type of the C language or a *type_name* defined beforehand in the 'C fragment for the definition of types for attributes' part. If specification of `inh` (inherited) or `synt` (synthesized) is omitted, it is assumed to be a synthesized attribute. Thus,

```
%terminal  ID  :
                name : NAMETYPE ;
```

specifies that terminal `ID` has a synthesized attribute `name` of type `NAMETYPE`. Terminals can have synthesized attributes but no inherited attributes.

If grammar symbols do not have attributes, we can only write grammar symbol names, for example,

```
%terminal DCL, USE ;
```

Moreover, if a nonterminal does not have attributes, we can omit the corresponding declaration, as in Yacc.

3.2.3.2 Declaration of equivalence classes of inherited attributes

The notion of equivalence classes is a characteristic feature of ECLR-attributed grammar. Each equivalence class is specified by the format

'%equiv' { *inherited_attribute_occurrence* || ',' }+ ';'

(*inherited_attribute_occurrence* will be described shortly.) For instance, in the same example,

```
%equiv  stlist.env, st.env ;                                  ( # )
```

specifies that two inherited attributes `stlist.env` and `st.env` are in the same equivalence class.

As can be seen here, an (*inherited_* or *synthesized_*) *attribute_occurrence* is expressed by the format

attribute_occurrence : *grammar_symbol* '.' *attribute_name*

For the present, any inherited attribute for which the user does not explicitly specify the equivalence class is assumed to form its own equivalence class with only one element.

We can also use a short-hand notation here. Its format is

'%equiv' '*' '.' *inherited_attribute_name* ';'

where * means a 'wild card'. In the above example (#), if `env` was an inherited attribute of only `stlist` and `st`, we could replace (#) by

```
%equiv  *.env ;
```

3.2.3.3 Declaration of precedence and associativity of grammar symbols

Precedence and associativity of grammar symbols (e.g. operators) can be specified in the same way as in Yacc, as follows.

```
%left ...
%right ...
%nonassoc ...
```

This is a purely syntactic feature which corresponds to the disambiguating rules for the LR grammar.

3.2.4 Syntactic and semantic rules

3.2.4.1 Syntactic rules

The specification of syntactic rules is the same as in Yacc. Readers are referred to [Johnson 75] for details. It is based on an almost pure BNF notation, and should fit into an LALR(1) grammar with disambiguating rules.

3.2.4.2 Semantic rules

Semantic rules are described in the part enclosed by '{' and '}' after each syntactic rule.

semantic rules :
'{' {*attribution rule* | *context condition* | *debugging statement*} '}'

Semantic rules (in general sense) can be divided into three categories in Rie:
(1) *attribution rules*,
(2) *context conditions*, and
(3) *debugging statements*.
They are explained in the following.

(1) Attribution rules

We call here the normal kind of semantic rule '*attribution rule*'. An example is

```
dclist.env = enter( dc.name, nullenv() ) ;
```

An *attribution rule* should be described in the format

attribution rule : *attribute_occurrence* '=' *expression* ';' (*)

We said before that an *attribute_occurrence* is written in the format

attribute_occurrence : *grammar_symbol* '.' *attribute_name*

But, if the same grammar symbol appears several times in one syntactic rule, their *attribute_occurrences* should be distinguished by indices [1], [2], [3], ... counted from the left, that is, an *attribute_occurrence* might also be in the form

attribute_occurrence : *grammar_symbol* '[' *index* ']' '.' *attribute_name*

An example of attribution rule using such *attribute_occurrences* is

```
stlist[2].env = stlist[1].env ;
```

In *expression* in the right hand side of (*) we can use any expression in the C language. Besides, in that *expression* we can use *attribute_occurrences* just like variables in the C language.

Let us see what conditions should the attribution rule (*) satisfy.
If the corresponding syntactic rule is of the form

X0 : X1 X2 .... Xn (* *)

the *attribute_occurrence* in the left hand side of (*) should be a synthesized attribute of X0 or an inherited attribute of X1, X2, ..., Xn.

Recall also that semantic rules must be in **Bochmann normal form**. That is, if the corresponding syntactic rule is of the form (**), an *attribute_occurrence* appearing in the right hand side *expression* of (*) should be an inherited attribute of X0 or a synthesized attribute of X1, X2, ..., Xn (or a local attribute which will be described later).

Naturally, (*) should satisfy the conditions for ECLR-AG shown in the previous chapter including that of L-attributed property.

Attribution rules corresponding to the same syntactic rule can be specified in any order.

Note that *attribution rules* (and also *context conditions*) are assumed to be functional or applicative. If the evaluation of the *expression* in the right hand side of (*) contains side-effect, the construction may not work in a way expected by the user.

A hint to the right hand side *expression* of (*). Since the right hand side of (*) must be an expression, we can not write control statements, like if-then-else or while statements. But for expressing a condition, we can use conditional expression of the C language, for example,

```
X.a = u == v ? b : c
```

which means

```
X.a = if u == v then b else c
```

It is always possible to express other control structures by calling a C function from the expression.

Although side-effects are not allowed (the result is not assured) in the *expression*, we could write a rather tricky expression using a sequence of expressions separated by ',' operator of the C language, like

```
X.a = ( temp = f(X1.a1), g(temp) )
```

Comments

Comments can be used at any point in the semantic rule, in the format

```
'/*' any letter except*/ '*/'
```

Short-hand notation

An attribution rule which just copies an attribute value from the right hand side to the left hand side is called 'copy rule' or 'transfer rule'. It is well known that copy rules often occupy more than half of attribution rules. Therefore, we have introduced short-hand notations for them.

Copy rules can be divided into three patterns.

(i) Distribution: an inherited attribute value is copied from the father to sons.

```
(ex.)    X0  :  X1  X2
              { X1.i = X0.i ;
                X2.i = X0.i ; }
```

(ii) Lift: a synthesized attribute value is copied from one son to the father.

```
(ex.)    X0  :  X1  X2
              { X0.s = X2.s ; }
```

(iii) Thread: values of inherited and synthesized attributes are copied like a 'thread', from the father to a son (normally oldest), from an older son to a younger son (possibly several times), and from a son (normally youngest) to the father.

```
(ex.)    X0  :  X1  X2  X3
              { X1.i = X0.i ;
                X2.i = X1.s ;
                X3.i = X2.s ;
                X0.s = X3.s ; }
```

All of the three cases can be expressed using a short-hand notation.

(i) and (ii): In these cases, we use the format

'%transfer' { *attribute_name* || ',' }+ ';'

```
(ex.)    X0  :  X1  X2  X3
              { %transfer i, j, s ; }
```

If i, j ∈ I(X0), i ∈ I(X1), j ∈ I(X2), i ∈ I(X3) and s ∈ S(X0), S(X3), this means

```
              { X1.i = X0.i ;  X2.j = X0.j ; X3.i = X0.i ;
                X0.s = X3.s ; }
```

(end of example)

In the '%transfer' notation, the following conditions should be satisfied.

- the nonterminal in the left hand side of the syntactic rule (i.e. X0) should have (all) referred attribute(s) (i.e. attributes referred to in '%transfer' notation), and
- for attribute(s) in the right hand side of the syntactic rule,
-- in case it is an inherited attribute, there should be at least one grammar symbol having the referred attribute, and
-- in case it is a synthesized attribute, there should be exactly only one grammar symbol having the referred attribute.

(iii): In this case, we use the format

'%thread'

{ *inherited_attribute_name synthesized_attribute_name* || ',' }+ ';'

```
(ex.)    X0  :  X1  X2  X3
              { %thread  i  s , j  t ; }
```

If i,j ∈ I(X0), s,t ∈ S(X0), i ∈ I(X1), s ∈ S(X1), j ∈ I(X2), t ∈ S(X2), i ∈ I(X3) and s ∈ S(X3), this means

```
{ X1.i = X0.i ;  X3.i = X1.s ;  X0.s = X3.s ;
  X2.j = X0.j ;  X0.t = X2.t ;  }
```
(end of example)

Note that not all nonterminals in the right hand side of the production have attributes i,s or j,t.

In the '%thread' notation, the following conditions should be satisfied.
- for each pair of inherited and synthesized attributes,
-- the nonterminal in the left hand side of the syntactic rule (i.e. X0) should have the referred pair, and
-- there should be at least one nonterminal (e.g. Xi) in the right hand side of the syntactic rule having the referred pair.

In using these short-hand notations, the user should use the same (*inherited_* or *synthesized_*) *attribute_names* for relevant attributes.

Sometimes, some attributes declared within '%transfer' or '%thread' statements will have their values determined by non-copy rules. This case can also be expressed using the '%except' statement, as follows.

(a) (a repetition of)
'%except' *attribute_occurrence* '=' *expression* ';'

or

(b) '%except' {*attribute_occurrence* || ','}+ ';'

with corresponding attribution rule(s)
attribute_occurrence '=' *expression* ';'

somewhere afterwards.

(ex.) Instead of writing

```
X0  :  X1  X2  X3
       { X1.i = X0.i    ;
         X2.i = f(X1.s) ;
         X3.i = X2.s    ;
         X0.s = g(X3.s) ; }
```

we can either write

(a)
```
{ %thread i s ;
  %except X2.i = f(X1.s) ;
  %except X0.s = g(X3.s) ;  }
```

or

(b)
```
{ %thread i s ;
  %except X2.i, X0.s  ;
   . . .
  X2.i = f(X1.s) ;
  X0.s = g(X3.s) ; }
```

Local attributes

Common subexpressions appear often in attribution rules, increasing the length of description. A simple example is

(ex.)
```
X0  :  X1  X2  X3
       { X1.i = X0.i ;
         X2.i = g(f(X1.s), X0.i ) ;
         X3.i = h(f(X1.s), X0.i ) ; }
```

In this example, f(X1.s) is a common subexpression. Besides the readability of description, this also affects efficiency of evaluation (compilation) since the same function will be evaluated twice.

Based on such consideration, we have introduced 'local attributes'. It is an attribute which is associated with a syntactic rule, instead of being associated with a grammar symbol. It is valid only in semantic rules (including conditions) associated with a syntactic rule. A local attribute is evaluated only once while processing the corresponding syntactic rule. Thus, it retains the single-assignment property or the applicative feature of attribute grammars.

In Rie, local attributes should be declared before using them. The declarations are as follows:

(possibly a plurality of)

'%local' *type_name* *local_attribute_name* '=' *expression* ';'

(ex.) The example above can be rewritten as

```
X0  :  X1  X2  X3
       { %local int f_val = f(X1.s) ;
         X1.i = X0.i ;
         X2.i = g(f_val, X0.i) ;
         X3.i = h(f_val, X0.i) ; }
```

Local attributes as well as ordinary attributes should satisfy the conditions of ECLR-attributed grammars mentioned in a previous chapter, including the L-attributed property.

Local attributes can also be used to avoid multiple occurrences of side-effect (although side-effects are not formally allowed). The following is taken from the specification of PL/0 by Rie.

(ex.)

```
statement : IF condition then1 statement
       { %local int ilab = genlab() ;
         %transfer I_env ;
         statement[1].code =
                concat(4, condition.code, gen(JPC,0,ilab),
                          statement[2].code, gen(LAB,0,ilab)) ;
       }
```

Here, `genlab()` is a function with side-effect to generate unique label name. By writing `genlab()` in the expression for a local attribute, it is called only once to give the necessary label.

We have had a result that using short-hand notations and local attributes, about 25% (in lines) was reduced in the syntactic and semantic part description of the Pascal semantic analyzer.

(2) Context condition

A *context condition* is described by the format

context condition : ['%condition' *expression*] (*)
'%message' *message* ';'

Here, *expression* means the context condition (context-sensitive or semantic condition) to be satisfied in the corresponding syntactic rule. It is described in the form of a (boolean) expression in the C language. *Attribute_occurrences* and *local_attribute_names* can also be used like variables in this *expression*, similarly to the *expression* in the right hand side of *attribution rules*. Note also that due to the Bochmann normal form, only *attribute_occurrences* which are allowed in the right hand side of *attribution rules* can be used in this *expression* (i.e. inherited attributes of the nonterminal of the left hand side of the syntactic rule and synthesized attributes of the grammar symbols in the right hand side of the syntactic rule are allowed).

If *expression* in (*) is evaluated to be 0 (i.e. false in the C language) the context condition is violated in the course of compilation, and *message* specified after '%message' is printed to standard output.

The *message* in (*) has the form of parameter(s) of 'printf' function in the C language. Normally, it is enclosed by " and ".

(ex.)

```
%condition lookup( dc.name, dclist[2].env )
%message   "*** ERROR: redeclaration of identifier" ;
```

We can also print several values of expressions with formatting specification, like in the 'printf' function.

```
%condition  ...
%message   "error at %d, ... is %d",  position+1, st.value ;
```

supposing that expression 'position+1' and *attribute_occurrence* 'st.value' is of integer type.

In the present implementation, a *context condition* is evaluated at reduction time of a syntactic rule, like synthesized attributes.

(3) Debugging statements

In the course of developing a compiler, we might need to check intermediate values of attributes etc. To help this, Rie supports a debugging statement. The format is

'%debug' '{' *statements* '}' ';'

where *statements* may be any list of statements in the C language, allowing *attribute_occurrences* in them.

(ex.) `%debug { printf( "sym is %s\n", ID.name ) ; ... } ;`

Whether debugging statements are effective or not can be specified at compile time (not at generation time) (see section 3.3.2).

3.2.5 C fragment for user-defined functions

In this part, user-defined functions which are called from semantic rules are described in the C language. We can also use the '`#include`' feature of the C language and put the actual program in other files, as well.

In addition to those, a lexical analyzer and a main program should be supplied. But since these two are a little bit special, they are explained below.

This 'C fragment for user-defined functions' part is simply copied to the generated compiler, like the 'C fragment for the definition of types for attributes' explained before.

3.2.5.1 Lexical analyzer

In Rie, the lexical analyzer should be supplied as a function named

`yylex()`

The name is made the same as the one generated by Lex.

The `yylex()` function should return

(i) token (terminal), and

(ii) attribute of token (if any)

when it is called.

(i) As for 'token', the `yylex()` function can just return the same *terminal_name* which is used in the description of syntactic rules, as in

`return` *terminal_name* ;

(ex.) If `yylex()` reads a string which corresponds to the terminal `ID`, it can just execute

`return ID ;`

(ii) As for the attribute of the token, the `yylex()` function must return it conforming to the following interface. If the terminal called *terminal_name* has an attribute called *attribute_name*, we assign to a variable whose name is a concatenation of

'`rrlval.rr`'*terminal_name* '.' *attribute_name*

Here, `rrlval` is a variable supplied by the generator.

(ex.) If the terminal `ID` has a synthesized attribute `name` as in

`%terminal ID : name : NAMETYPE ;`

we assign the attribute value as follows.

`rrlval.rrID.name = ... ;`

Thus, the function `yylex()` will normally look like

```
yylex()
{ ...
 switch( input_character )
  ...
 case c :
      rrlval.rrterminal_name.attribute_name = ... ;
      return terminal_name ;
  ...
}
```

We can also have the same effect using Lex. In that case, input specification to Lex will, for the above example of `ID`, look like

```
[A-Za-z] [A-Za-z0-9]*     { ...
                            rrlval.rrID.name =  ... ;
                            return ID ;
                          }
```

3.2.5.2 Main program

In the compiler generated by Rie, the syntax analyzer drives the whole compiler. So, the major role of the main program is to call the syntax analyzer. Its template is as follows.

```
main( argc, argv )
  int argc ;
  char *argv[] ;
  {
      rroption( argc, argv )   /* get compile-time options */
       ...                     /* initialization */
      yyparse( argc, argv ) ;  /* call the syntax analyzer */
       ...                     /* finalization */
  }
```

The parameter `argc` and `argv` is to get compile-time options (see section 3.3.2). `yyparse()` is the syntax analyzer.

3.3 Use of the Rie system (generation time and compile time options)

Here, we explain how to actually use Rie, including options.

In the following, 'generation time' means the time when Rie reads a specification and generates a compiler, and 'compile time' means the time when the generated compiler runs.

3.3.1 Generation-time

Suppose that (the generation system of) Rie can be invoked as a command named '`rie`' (how to define such a command depends on the operating system). Then, the format of the command for invoking Rie at generation-time is

'`rie`' { '`-l`' | '`-v`' | '`-m`' | '`-d`' } *specfile*

where `-l`, `-v`, `-m`, `-d` represent generation-time options, *specfile* is the file where the specification of the language or compiler is written.

(ex.) `rie -l -d -m pl0.r`

Rie outputs the compiler in the file called '`rie.tab.c`'.

The meaning of each generation-time option is as follows.

`l`: List *specfile* to the file '`rie.lis`'. If errors in specification format etc. occur in *specfile*, error messages are printed at the line where errors were found. This option is useful to correct specification format errors occurring in *specfile*.

`v`: This option has the same function as the '`v`' option in Yacc. It outputs information about LR states and their transition relations etc. used in LALR(1) syntax analysis. The output is the file '`yacc.out`'. This option is useful to check syntactic rules of *specfile* if there are shift/reduce conflicts or reduce/reduce conflicts etc.

`m`: This option can be used when the attribute grammar given in *specfile* violates ECLR-attributed property. It shows which LR states and which semantic rules violate the condition. The output is the file '`rie.out`' and its format is similar to that of the '`v`' option.

`d`: This option outputs information about the internal number of terminals and how attributes are allocated into attribute stacks. The output is the file '`rie.def`'.

We can summarize output files and related options as follows.

```
        riepar.  (skeleton. cf. 3.5)
          |                        ---------> rie.tab.c   (compiler)
          |                -l      ---------> rie.lis
specfile  --> rie -->      -v      ---------> yacc.out
                           -m      ---------> rie.out
                           -d      ---------> rie.def
```

3.3.2 Compile-time

Rie outputs into the file 'rie.tab.c' the source program of the compiler written in the C language. Therefore, we must call a C compiler to make an executable form. This can be made, for example in Unix, by

```
cc    rie.tab.c -o compiler
```

(in some system, we also need to 'link' as in VAX/VMS).

Then, we can execute the compiler obtained which is in executable form. Suppose that the executable compiler can be executed by the command compile .

Options are also supplied at compile-time, as well as at generation-time. We can invoke the compiler with compile-time options in the following format.

```
'compile' { '-y' | '-r' }
```

(ex.) The next is an example in Unix. '<' means redirection of input.

```
compile  -y  < sourcefile
```

Each option means

y: Trace. Each time the syntax analyzer reads a token, it displays information like transition of LR states etc. and show how compilation proceeds.

r: Make the '%debug' part (section 3.2.4(3)) in the semantic rule effective. Without this option, the debugging part is normally neglected.

3.3.3 Connecting Rie with Lex in Unix

Usual command sequence for making a compiler using Rie and Lex is as follows.

```
lex     lexspecfile.l
rie     riespecfile.r
cc      rie.tab.c  -ll
a.out   <  sourcefile
```

See also the example in the next section.

3.4 A complete example

A complete example in Unix is shown below. This corresponds to the example attribute grammar given in chapter 2, and combines Rie with Lex.

(1) file `dcuse.l`

```
%%
[ \n]                    { ECHO ; /* skip space and newline */ }
dcl                      { ECHO ;  return DCL ;  }
use                      { ECHO ;  return USE ;  }
[A-Za-z][A-Za-z0-9]*     { char *p ; ECHO ;
                           p = (char *)malloc( strlen(yytext)+1 ) ;
                           strcpy( p, yytext ) ;
                           rrlval.rrID.name = p ;
                           return ID ; }
.                        { ECHO ; return yytext[0] ; }
```

(2) file `dcuse.r`

```
%{
          /* type definition */
#define  ENVTABLESIZE   100
#define  NULLENV         -1
#define  NOTFOUND        -1
typedef  int     ENVTYPE ;
typedef  char    *NAMETYPE ;
typedef  int     ENTRYTYPE ;
NAMETYPE envtable[ENVTABLESIZE] ;
%}
          /* symbol and attribute declaration */
%nonterm program ;
%nonterm dclist :
            env : ENVTYPE ;
%nonterm dc :
            name : NAMETYPE ;
%nonterm stlist, st :
            env : ENVTYPE inh ;
%terminal ID :
            name : NAMETYPE ;
%terminal DCL, USE ;
%equiv    stlist.env, st.env ;

%%        /* syntax and semantic rules */
program   : dclist stlist
          { stlist.env = dclist.env ; } ;
dclist    : dclist dc
          { dclist[1].env = enter( dc.name, dclist[2].env ) ;
            %condition lookup( dc.name, dclist[2].env ) == NOTFOUND
            %message   "*** ERROR: redeclaration of identifier" ;
          } ;
dclist    : dc
          { dclist.env = enter( dc.name, nullenv() ) ; } ;
dc        : DCL  ID ';'
          { /* dc.name = ID.name ; */
            %transfer name ;
          } ;
stlist    : stlist st
          { /* stlist[2].env = stlist[1].env ;
               st.env        = stlist[1].env ; */
            %transfer env ;
          } ;
```

```
stlist    : st
          { /* st.env = stlist.env ; */
            %transfer env ;
          } ;
st        : USE ID ';'
          { %condition lookup( ID.name, st.env ) != NOTFOUND
            %message   "*** ERROR: identifier not declared" ;
          } ;

%%        /* user-defined functions */
#include <stdio.h>
#include "lex.yy.c"
ENVTYPE nullenv ()
  {  return -1 ; }
ENVTYPE enter( name, env )
  NAMETYPE name ;
  ENVTYPE env ;
  {  envtable[ ++env ] = name ;
     return env ;  }
ENTRYTYPE lookup( name, env )
  NAMETYPE name ;
  ENVTYPE env ;
  {  int i ;
     for( i = 0 ; i <= env ; i++ )
         if (!strcmp( name, envtable[i] )) return i ;
     return NOTFOUND ;  }
main( argc, argv )
  int argc ;
  char *argv[] ;
  {  rroption( argc, argv ) ;
     yyparse() ; }
```

(3) file `dcuse.dat`

```
dcl x ;
dcl y ;
dcl x ;
use x ;
use z ;
use y ;
use y ;
```

(4) command sequence

```
lex dcuse.l
rie dcuse.r
cc rie.tab.c -ll
a.out < dcuse.dat
```

(5) result

```
dcl x ;
dcl y ;
dcl x ;*** ERROR: redeclaration of identifier

use x ;
use z ;*** ERROR: identifier not declared

use y ;
use y ;
```

3.5 Installation guide

(This section is subject to change.)

The original version of Rie currently runs on Sun3 and Sun4 under Sun-OS (Unix 4.2/4.3 BSD). The distribution tape is based on this version.

Rie can be installed as follows.

(1) Load down from the streamer tape

The format of the streamer tape is

standard 'tar' format

blocksize = 20

Load down the contents of the tape by executing the following commands.

```
cd    directory_name_under_which_Rie_is_to_be_created
tar   vx
```

(2) Create the Rie system

The tape contains only the source program of Rie. So, we have to make the executable file.

First, execute

```
cd    rie/src
```

Next, if we can use the 'make' command, execute only

```
make
```

If not, execute

```
cc   r[yr]?.c   -o   rie
```

and compile 11 files in total which are ry1.c ... ry4.c and rr1.c ... rr7.c .

This makes the Rie system.

Copy the file 'riepar.' into '/usr/local/lib/rie/riepar.' (The position is defined by '#define PARSER "/usr/local/lib/rie/riepar." in files.h, and it can be modified.)

(3) Test use of Rie

The distribution tape includes also several sample specifications for Rie, in addition to the source program of Rie. Let us generate the PL/0 compiler, for example.

Before that, it might be better to arrange in a way that the Rie system can be invoked simply by typing the command 'rie'. Usually, this can be made in Unix by 'alias'ing or by setting a 'path' (e.g. set path = ($path /rie/src)) for the executable form of Rie.

First, change the current directory by

```
cd     ../exm/pl0
```

Then, execute the following command(s).

(if 'make' can be used)

```
make   pl0
```

(if 'make' can not be used)

```
rie    pl0.r
mv     rie.tab.c   pl0.c
```

Then, the source program of the compiler for PL/0 will be generated in the file 'pl0.c'. Now, execute

```
cc     pl0.c    -o   pl0
```

and make the load module pl0. Next, run the PL/0 compiler by

```
pl0 < ex1.pl0
```

4. Concluding remarks

In this manual, we have described an introduction to attribute grammars, LR- and ECLR-attributed grammars, and then we have presented a user's manual for the Rie system which is a compiler generator based on ECLR-attributed grammars.

Any questions, comments and suggestions are welcome. Please contact the following.

Masataka Sassa
University of Tsukuba
Institute of Information Sciences and Electronics
Tennoodai, Tsukuba-shi, Ibaraki-ken, 305, Japan

tel: 0298-53-5511, 5051 (International +81-298-53-5511, 5051)
FAX: 0298-53-5206 (International +81-298-53-5206)
E-mail: sassa@is.tsukuba.ac.jp

Acknowledgments

The authors wish to thank Rie Inada and other members of our laboratory for assistance in the development of the Rie system, and Jorma Tarhio of the Univ. of Helsinki for his helpful comments on the report.

References

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

[BSI 82] BS 6192: Specification for computer programming language Pascal, British Standards Institution, 1982.

Farrow, R.: Generating a Production Compiler from an Attribute Grammar, IEEE Software, Vol.1, No.4, pp.77-93, 1984.

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

Johnson, S.C.: Yacc - Yet Another Compiler Compiler, Computer Science Tech. Rep. 32, AT&T Bell Lab., 1975.

Jones, N.D. and Madsen, M.: Attribute-influenced LR Parsing, Lecture Notes in Comp. Sci., Vol.94, pp.393-407, 1980.

Kastens, U., Hutt, B. and Zimmermann, E.: GAG: A Practical Compiler Generator, Lec. Notes in Comp. Sci., Vol. 141, Springer, 1982.

Knuth, D.E.: Semantics of Context-Free Languages, Math. Syst. Th., Vol. 2, No. 2, pp. 127-145, 1968, correction *ibid.* Vol. 5, No .1, pp. 95-96, 1971.

Koskimies, K., et al.: The Design of the Language Processor Generator HLP84, Dept. of Comp. Sci., Univ. of Helsinki, Rep. A-1986-4, 1986.

Lesk, M.E.: Lex - A Lexical Analyzer Generator, Computer Science Tech. Rep. 39, AT&T Bell Lab., 1975.

Nakata, I. and Sassa, M.: L-attributed LL(1) Grammars are LR-attributed, Inf. Process. Lett., Vol. 23, No. 6, pp. 325-328, 1986.

Reps, T. and Teitelbaum, T.: The Synthesizer Generator Reference Manual, 3rd ed., Springer, 1989.

Sassa, M. and Ishizuka, H.: Description of PL/0 Compiler by ECLR-attributed Grammars (in Japanese), Tech. Memo PL-2, Inst. of Information Sciences, Univ. of Tsukuba, 1984.

[Sassa et al. 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 et al. 85b] Sassa, M., Ishizuka, H. and Nakata, I.: A Contribution to LR-attributed Grammars, J. Inf. Process., Vol. 8, No. 3, pp. 196-206, 1985.

Sassa, M.: Attribute Grammars, a Tutorial (in Japanese), Computer Software (Journal of Japan Society of Software Science and Technology), Vol. 3, No. 4, pp. 73-91, 1986.

Sassa, M., Ishizuka, H. and Nakata, I.: ECLR-attributed Grammars: a Practical Class of LR-attributed Grammars, Inf. Process. Lett., Vol. 24, pp. 31-41, 1987.

Yamashita, Y., Sassa, M. and Nakata, I. : A Friendship Club Problem and its Applications to Attribute Grammars (in Japanese), Computer Software, Vol. 4, No. 3, pp. 28-40, 1987.

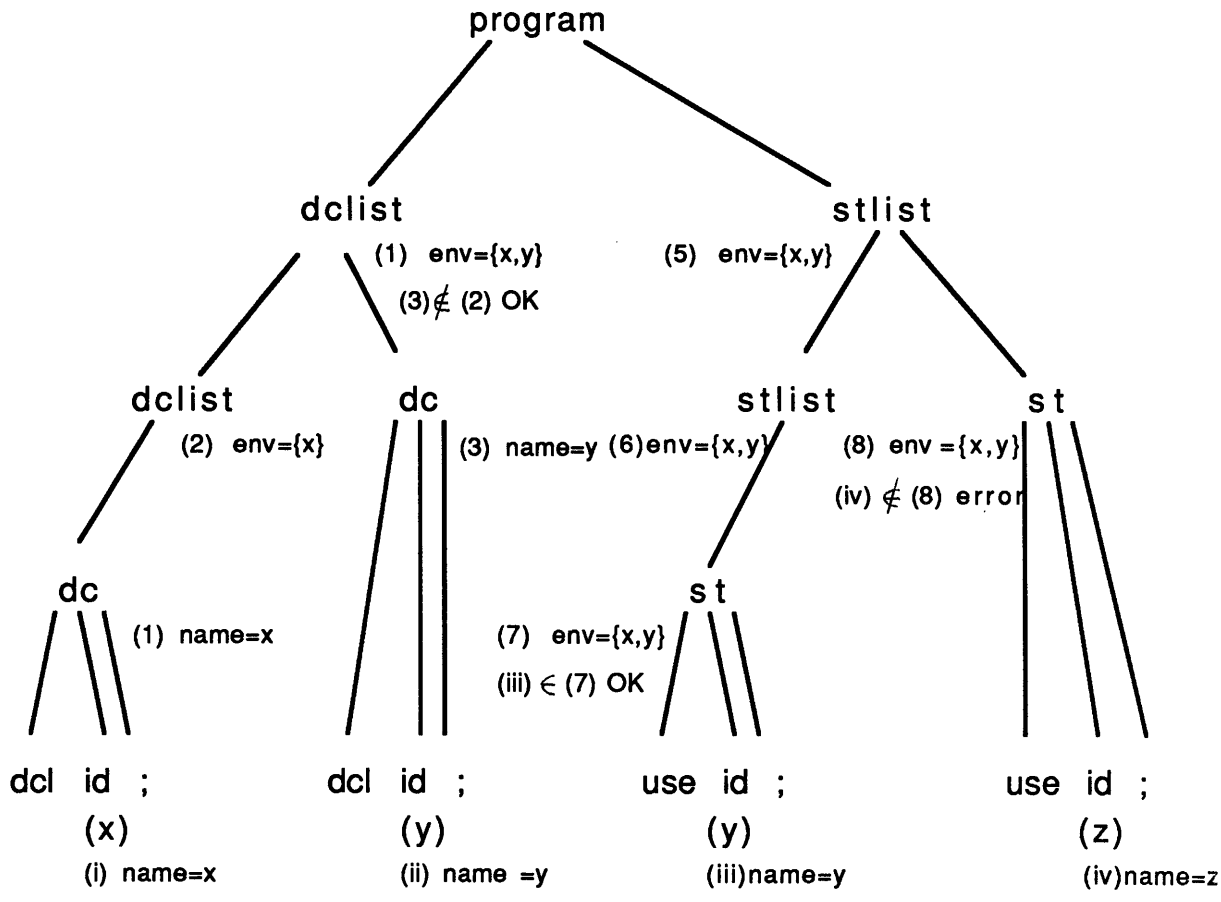


Fig. 1 Parse tree and attributes for the source program (S2)

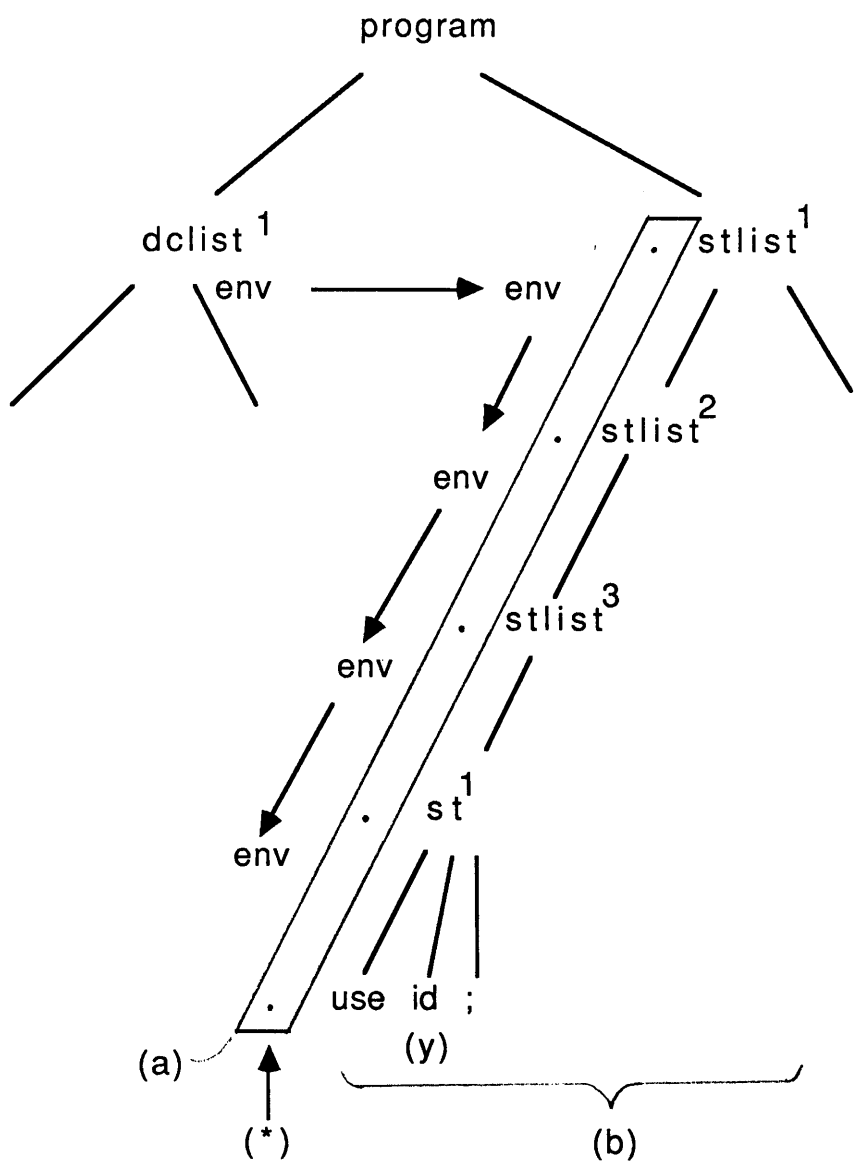
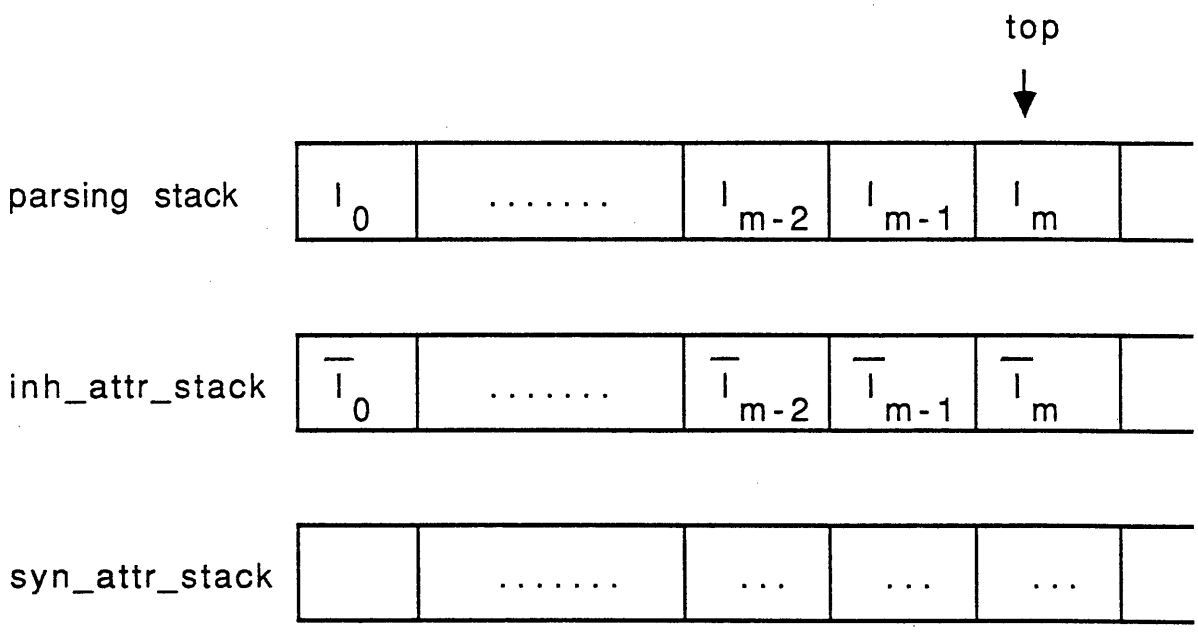
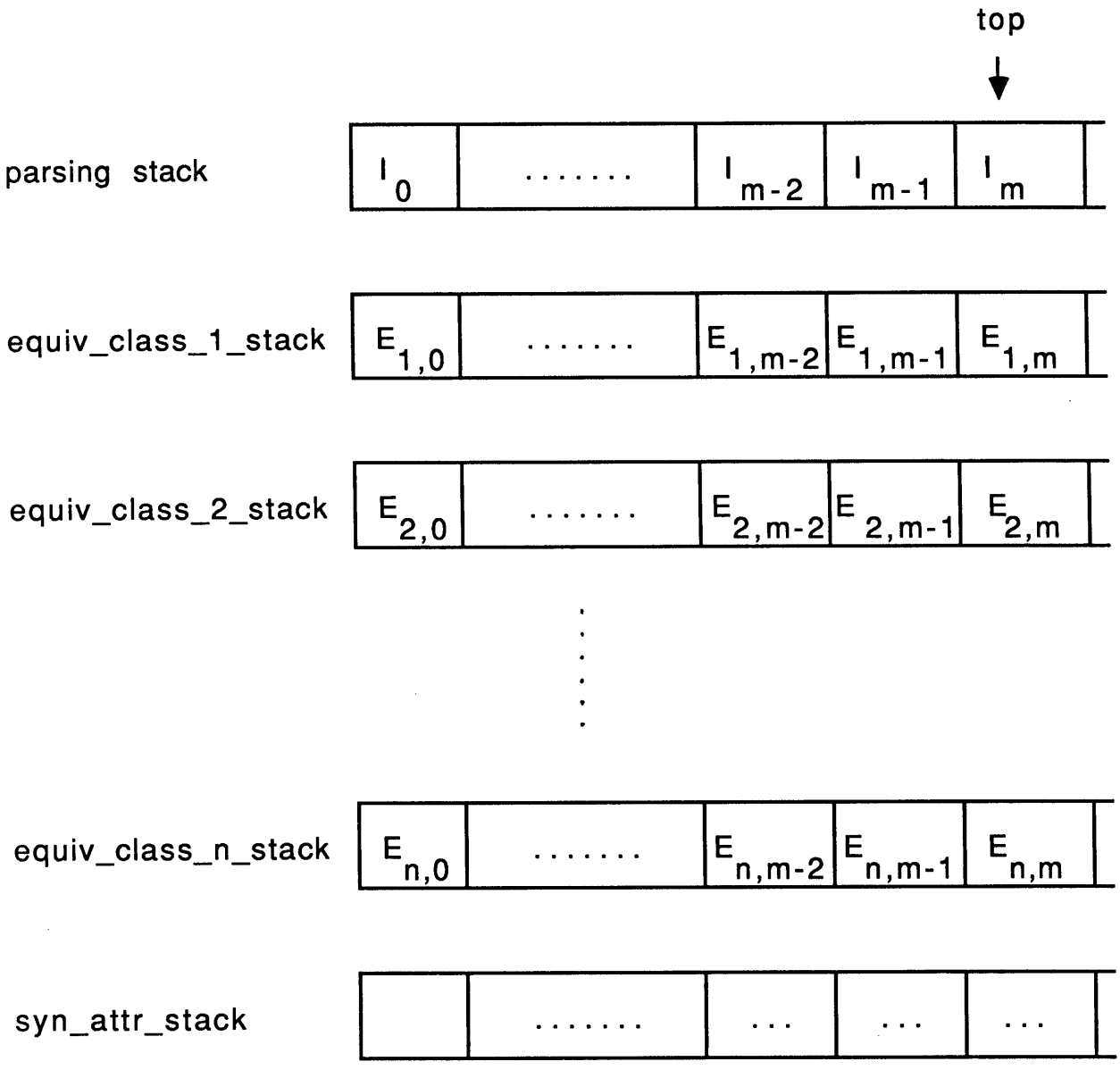


Fig. 2 Snapshot of LR parsing



I_i is an LR state, $\overline{I}_i$ is a record containing values of inherited attributes in IN (I_i) .

Fig. 3 Organization of stacks for analysis of LR-AG



$E_{i,m}$ contains the values of inherited attributes in $EC_i \cap IN(I_m)$

Fig. 4 Organization of stacks for analysis of ECLR-AG

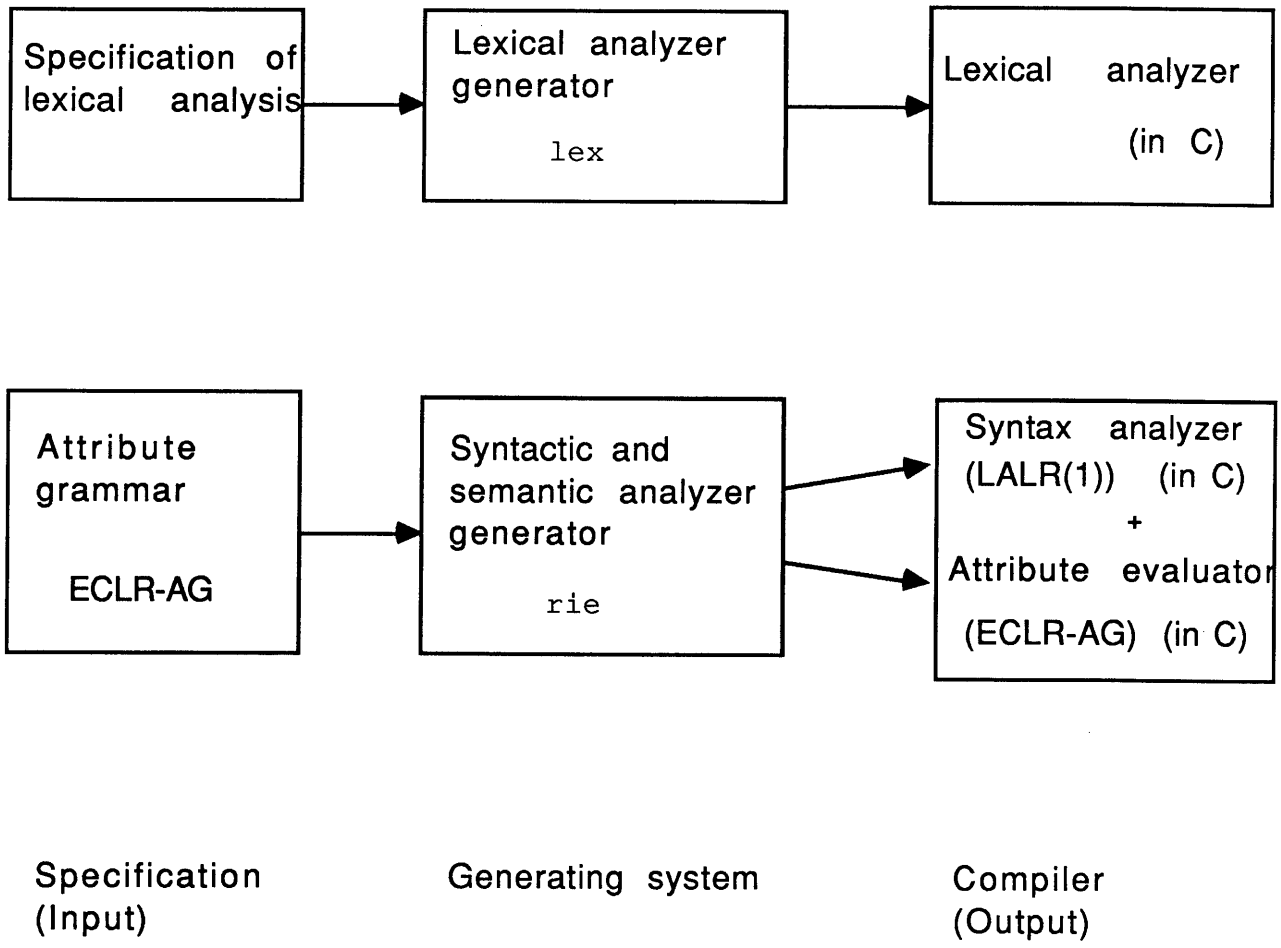


Fig.5 Organization of the system

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

REPORT DOCUMENTATION PAGE	REPORT NUMBER ISE-TR-90-82

TITLE

Rie - Introduction and User's Manual

AUTHOR(S)

Masataka Sassa, Harushi Ishizuka, Motoki Sawatani,
and Ikuo Nakata

(Institute of Inf. Sciences, Univ. of Tsukuba)

REPORT DATE	NUMBER OF PAGES
June 20, 1990	26 + 5 figs
MAIN CATEGORY	CR CATEGORIES
Programming Languages - Processors	D.3.4, D.3.1, D.2.3

KEY WORDS

Compiler Generators, Attribute Grammars

ABSTRACT

This report describes Rie, a compiler generator based on attribute grammars.

In the first part, the concept of attribute grammars is presented with an example. Then, the class of LR-attributed grammars is introduced. It is a class of attribute grammars for which attributes can be evaluated in a single pass during LR parsing. The ECLR-attributed grammar is defined by incorporating equivalence classes to the LR-attributed grammar.

Rie is a compiler generator based on ECLR-attributed grammars. From user's viewpoint, it can be regarded as an extension of Yacc. The syntax analysis part of the generated compiler is based on LALR(1) grammar with disambiguating rules. The base language of the semantic part is C. In the second part, the manual of Rie is presented, together with the description format, operating instruction, and a complete example.

SUPPLEMENTARY NOTES