Rie and Jun:
Towards the Generation of all Compiler Phases [*]

Masataka Sassa

Institute of Information Sciences and Electronics, University of Tsukuba
Tsukuba-shi, Ibaraki-ken, 305, Japan
sassa@is.tsukuba.ac.jp

Abstract

Two compiler generators, both based on attribute grammars, have been used together in an attempt to generate almost all compiler phases.

Rie is a compiler generator based on a one-pass attribute grammar called ECLR-attributed grammar [Sassa 87]. The generated compiler evaluates attributes in parallel with LR parsing. It can be used to generate one-pass compilers or the front-end of multipass compilers.

Jun is a compiler generator which works on tree grammars. It is based on finitely recursive attribute grammars, a class of attribute grammars which allow circularities in attribute dependency [Farrow 86]. It overcomes the difficulty of circularities in the attribute dependency which often appear in data-flow equations of optimizers. The evaluator generated by Jun evaluates attributes on a tree, rather than a source program. This same formalism can be used to generate code generators and interpreters. So, the single Jun system can cover most of the compiler back-end.

By combining both Rie and Jun, the generation of almost all compiler phases may be possible.

1. Introduction

Recently there have been many advances in the automatic generation of compilers. The major reason for these advances seems to be the study of efficient evaluators based on attribute grammars (AGs) [Knuth 68]. Many compiler generators have been developed using AGs, for example MUG2 [Ganzinger 82], GAG [Kastens 82], Linguist-86 [Farrow 84], and

[*] This work was partially supported by the University of Tsukuba.

HLP84/TOOLS [Koskimies 88]. AGs have been usually applied to the generation of compiler front-ends, i.e. parsers, semantic analyzers, and sometimes code-generators [Ganapathi 85] (affix grammars), but they have not been applied to the generation of compiler back-ends or interpreters in a natural way except for few systems (cf. [Ganzinger 82]). From the software engineering viewpoint of compiler writers, it is better to handle most compiler phases with a smaller concept.

Based on the above consideration, we tried to use AGs also for the specification and generation of language-based editors, compiler back-ends, and interpreters. This paper presents an attempt to generate almost all compiler phases on the formalism of attribute grammars.

Overall Structure of the Compiler Development Environment

Our long range plan is to generate an integrated programming environment using a small set of generators. The overall structure of the compiler development environment is shown in Fig. 1.

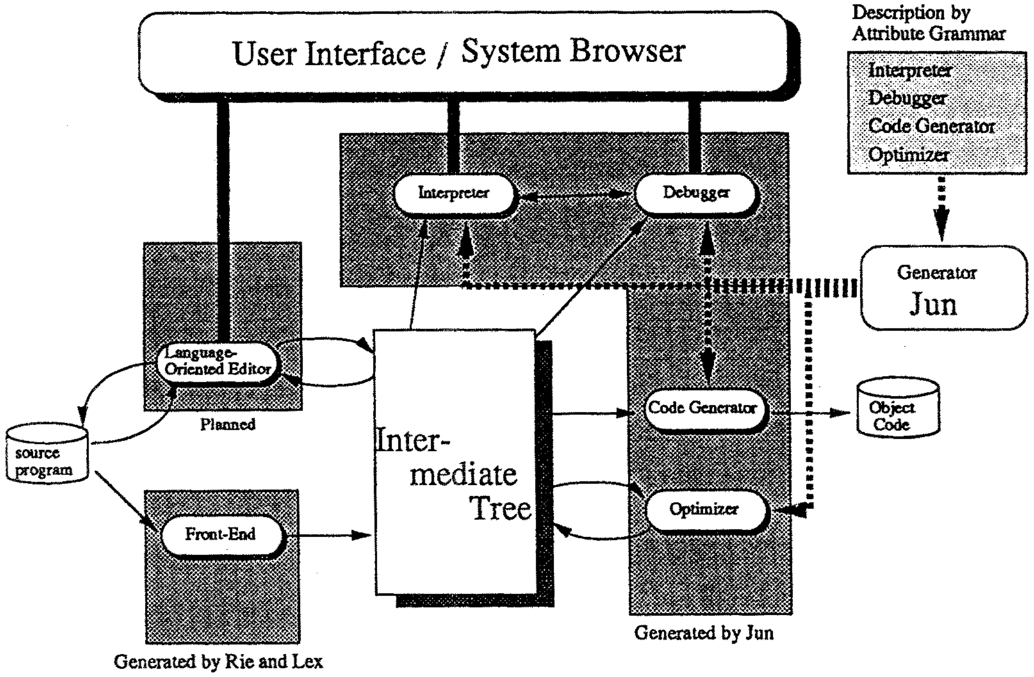


Fig. 1 Overall Structure of Compiler Development Environment

The front-end of a compiler is generated by Rie and Lex [Lesk 75]. The front-end converts the source program into an intermediate tree. Rie reads in the specification of the source-to-tree translation, which is written in a class of one-pass AGs called ECLR-attributed grammar, and generates an attribute evaluator which evaluates attributes during LR parsing. Rie is written in C and generates C programs. Rie can also be used to generate one-pass compilers.

A major problem in applying AGs to the back-end of compilers reside in that usual formalism for some of these phases involves iteration. That is, in optimizers, there is a difficulty of circularities in attribute dependency which often appear in data-flow equations. A similar problem arises in interpreters, where loop statements force iterative evaluation of attributes.

To settle these problems, we used the *finitely recursive attribute grammars* proposed by Farrow [Farrow 86]. This is a class of AGs which allows circularities in attribute dependency. It is defined as an extension of the class of absolutely noncircular AGs. Broadly speaking, if there are cycles of attribute dependency but if there exists a least fixed-point (a minimum solution if it is a set) for each attribute in the cycle, then the evaluator can compute the attribute values using successive approximation. The evaluator generated by Jun does this. In addition, some other techniques for allowing iterative evaluation in interpreters have practically settled the problem of run-time loop statements.

All the back-end phases can be realized to process a common intermediate tree, which is in the form of an abstract syntax tree. Jun is based on tree grammars, and the generated evaluator can evaluate attributes on the intermediate tree, rather than a source program. Thus, the single Jun system can generate most of the back-end phases, i.e. data-flow analysis component of optimizers, code-generators and interpreters/debuggers, based on the same framework. For the moment, Jun is made as a prototype. So, it is written in Common Lisp (KCl) and generates programs in Common Lisp, for the ease of development.

The code-generator, interpreter, and the data-flow analysis component of the optimizer for a small language PL/0 have been made using Jun. The transformation component of the optimizer is still to be investigated. The generation of a language-oriented editor has been designed [Sassa 88] but not yet implemented. A visual debugger is currently under development. A debugger for the compiler writer (not the user of the language) to debug AG description is also planned based on the algorithmic debugging approach [Shapiro 83]. The generator of user interface, with multiple windows, buttons and menus, is partially complete.

In the following we mainly present the generation of compiler front-ends and back-ends using Rie and Jun.

2. Rie and Generation of Front-end

Rie [Sassa 90] is a compiler generator based on a class of one-pass AGs called ECLR (equivalence class LR)-attributed grammar [Sassa 87]. The ECLR-AG is a variant of LR-AG, where attributes can be evaluated during LR parsing [Sassa 85]. Equivalence classes are introduced to reduce space requirements. Rie can be used to generate one-pass compilers (from semantic analyzers to code generators) or the front-ends of multipass compilers.

Rie can deal with inherited attributes as well as synthesized attributes. One may wonder why we can use inherited attributes during bottom-up parsing. The key idea is that an LR state is made so that it contains all the possibilities of a syntax tree at the point of parsing. So even the parser is looking at a terminal of the input program, all the possible syntax trees which may come above that terminal can be foreseen [Sassa 85].

A resulting characteristic feature is that attribute evaluation can take place not only at reduction-time of the LR parser but also at state transition time. For example, in a syntax and semantic rule like

$$X \rightarrow X0\ X1\ X2$$
$$\{\ X1.env = X.env\ ;\ \}$$

the evaluation of inherited attribute X1.env is made when the LR parser enters a state including the LR item $[\ X \rightarrow X0 \cdot X1\ X2\ ]$. That is, inherited attributes of a nonterminal in the midst of the right hand side of a production are evaluated at the time parsing proceeds to the point of that nonterminal. This is in contrast with the usual bottom-up syntax-directed translator, where only synthesized attributes are allowed and evaluated at reduction-time.

An example AG description to be input to Rie is shown in Fig. 2.

```
. .. .
%nonterm block:                                                    }
      I_env:    envptr     inh,                                    }(a)
      tree:     treeptr    synt;                                   }
...
%equiv  constdefpart.I_env, constdeflist.I_env,                    }
        ...                                                        } (b)
        condition.I_env, ident.I_env;                              }
...
%%
...
block : constdefpart vardeclpart procdeclpart statement           ...(c)
      { %thread I_env S_env;                                   ...(e) }
        %except constdefpart.I_env = newenv(block.I_env);             }(d)
        block.tree = concat6(" (block ",                              }
                             constdefpart.tree,                       }
                             vardeclpart.tree,                        }
                             procdeclpart.tree,                       }
                             statement.tree,                          }
                             ")" );            } ;                    }
```

Fig. 2 Rie description for PL/0 (part)

Here, (a) shows the declaration of attributes and their types with the distinction of 'inherited' and 'synthesized', (b) shows that attributes `constdefpart.I_env`, `constdeflist.I_env`, etc., are in the same equivalence class (which roughly means that their storage can be shared in the attribute stack), (c) is a production, and (d) is the semantic rules associated with the production. Rie description allows for short-hand notations, such as abbreviating copy-rules and a '*thread*' of attributes. A thread (`%thread`, Fig. 2(e)) is a list of attributes whose values are passed like a "chain", normally by copying or by adding new information (`%except`). Rie also allows *local attributes* associated with a production rather than a nonterminal, which can be used to store temporary results of computation. These temporary results can then be used by more than one set of semantic rules.

The first version of Rie was made more than five years ago, and Rie has been applied to generate the semantic analyzer of ISO Pascal, a translator of a stream-based language Stella [Kuse 86], a compiler of a programming language called Gramp which is based on coupled context-free grammars [Yamashita 88], a compiler of neural network language, etc. Rie is designed and implemented as an efficient system. It is written in C and generates a parser and an attribute evaluator in C.

Generation of Front-end

The front-end of a compiler can be generated using Lex [Lesk 75] for the lexical analyzer and Rie for the syntactic and semantic analyzer.

In our compiler development environment, we use Rie to translate the source program into the internal tree which is in the form of an abstract syntax tree. Since Rie is based on C (to realize fast evaluation) and Jun on Lisp (for the ease of development), the front-end generated by Rie writes strings of S-expression of Lisp corresponding to the internal tree. An example of the generated S-expression for a source program is shown in Fig. 3.

```
;var  x;
;begin
;     x:=1+2
;end.
(pl0  (block (no_constdef) #1=(vardecl (name x))
(no_procdecl)(assign (var_id #1#)(plus (num 1)(num 2)))))
```

Fig. 3 Intermediate tree (S-expression) for a source program

3. Jun

Jun is a compiler generator based on a class of AGs called *finitely recursive attribute grammars* [Farrow 86]. Early formalism that used AGs had difficulty in specifying the data-flow analysis of optimizers, due to the circularities in attribute dependency which may often arise based on the recursive nature of data-flow equations [Babich 78]. It is usually possible to rewrite the semantic rules to make the attribute dependency cycle-free, but the rewritten semantic rules are not natural nor easily understandable. The evaluator generated by Jun can deal with such circularities. The details will be given shortly. This approach is different from previous works [Ganzinger 82][Lipps 89].

Jun can also generate interpreters which use some techniques to escape from circularities of attribute dependency arising in run-time loops (to be described later). Jun can of course generate usual evaluators without circular attribute dependencies, such as code generators. Thus it should be noted that the single Jun system can generate most of the compiler back-end, with the exception of tree transformation component of the optimizer. A prototype back-end (data-flow analysis component of optimizer, code generator, and interpreter) of a compiler for a small language PL/0 has been implemented.

Since Jun is presently a prototype, it is written in Common Lisp (KCl) and generates attribute evaluators in Common Lisp. Use of Lisp is mainly for the ease of data structure handling and early development.

Finitely Recursive Attribute Grammar and its Evaluator

The outline of the finitely recursive attribute grammar and its evaluator is as follows. See Fig. 4 for an example.

```
(1) S → S₁ ';' S₂
        { S₂.out = S.out ;
          S₁.out = S₂.live ;
          S.live  = S₁.live ; }
(2) S → ε
        { S.live = S.out ; }
(3) S → id ':=' E
        { S.live = E.use ∪ (S.out - {id.name}) ; }
(4) S → 'if' E 'then' S₁ 'else' S₂
        { S₁.out = S.out ;
          S₂.out = S.out ;
          S.live  = E.use ∪  S₁.live ∪  S₂.live ; }
(5) S → 'while' E 'do' S₁
        { S₁.out = E.use ∪ S.out ∪ S₁.live ;
          S.live  = E.use ∪ S.out ∪ S₁.live ; }

S.live: set of variables that are live on entry to S
S.out:  set of variables that are live on exit from S
E.use:  set of variables whose values at the entry of E are used in E
```

Fig.4 Attribute grammar G1 for live variable analysis

This formalizes the live variable analysis for a small language. To check whether the given AG is finitely recursive, first, make the attribute dependency graph for each production of G1 of Fig. 4 (see Fig. 5 (a)). Then take its closure and make the extended attribute dependency graph (see Fig. 5 (b)).

There is a cycle in the extended attribute dependency graph DG5*. The cycle consists of S_1*.out* and S_1*.live*. The set of nodes (attribute occurrences) in such a cycle is called a *circular dependency class* (*CDC*). Nodes x and y of an extended dependency graph belong to the same CDC iff x depends on y and y depends on x. A usual attribute not in a cycle will be in a CDC by itself. The CDC that contains an attribute occurrence x is denoted by $[x]$. By regarding each CDC as a single node, we can get an extended attribute dependency graph without cycles (see Fig. 5 (c)). This suggests that the evaluation of attributes in a CDC can be performed by successive approximation.

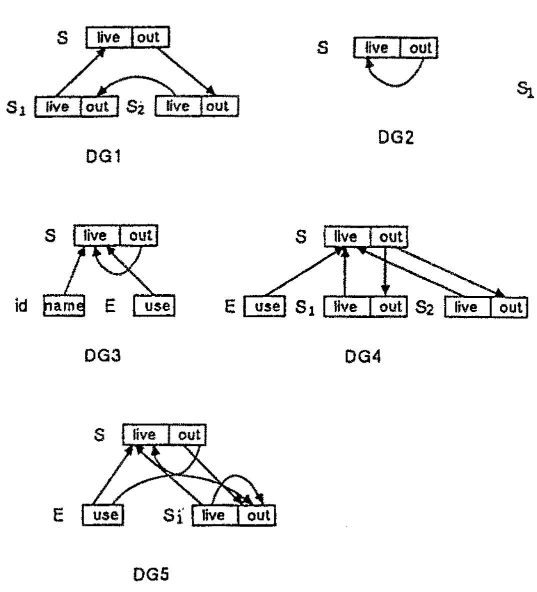
A finitely recursive attribute grammar is defined in the following way: let ATTRIBS be the set of attributes in the cycle of the dependency graph and FUNCTS be the set of corresponding semantic functions for these attributes. An AG is a *finitely recursive attribute grammar* iff:
1. The domain of all attributes in ATTRIBS constitutes a complete partial order (c.p.o.), in which it is possible to test pairs of elements for equality, and
2. All functions in FUNCTS are monotonic and converge (an ascending chain condition), that is,

$$f(s[0]) < f(s[1]) < \dots\dots < f(s[k]) = f(s[k+1])$$

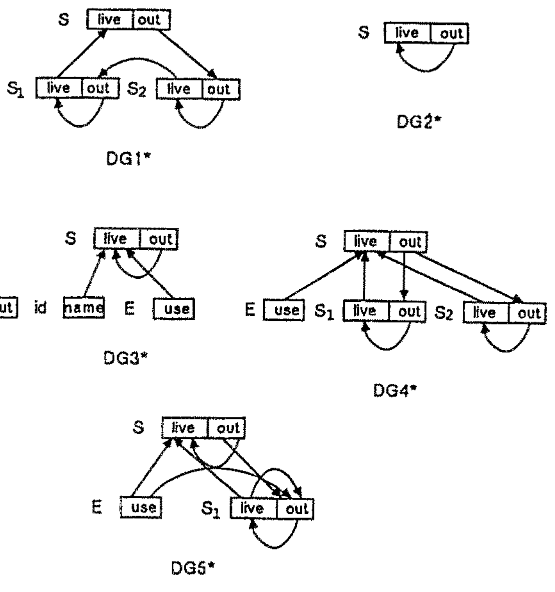
for

$$s[0] < s[1] < s[2] < \dots\dots$$

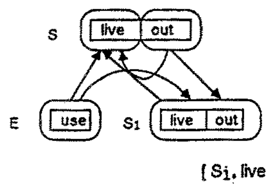
where $f \in$ FUNCTS, $s \in$ ATTRIBS, i of s[i] is the count of iteration. Here, $f(s[k]) = f(s[k+1])$ is called the *least fixed-point*.



(a) attribute dependency graph for each production



(b) extended dependency graph for each production



(c) CDC of production (5) of G1

Fig. 5 Attribute dependency of grammar G1

(We used an ascending chain condition and least fixed-point because the operator was '$\cup$' in G1. If the operator were '$\cap$', we could as well use a descending chain condition and greatest fixed-point .)

The finitely recursive attribute grammar is well-defined and it is an extension of the absolutely noncircular attribute grammar.

The above condition states that the values of attributes in cycles can be computed via successive approximation in a finite number of iterations. There may be several ways to make the evaluator. In Jun, we used Farrow's recursive synth-function evaluator [Farrow 86]. This attribute evaluator is static in the sense that the order of evaluation at each node of the tree is determined at generation time. For each synthesized attribute of each *class* a function is generated. An example of such a function *R-{S.live}* for the attribute *S.live* of the 'while statement' (production (5) of Fig. 4 or Fig. 5(c)) is shown in Fig. 6. Here the computation of $S_1.out$ and $S_1.live$ is made by successive approximation in the '**repeat**' statement.

```
R-{S.live} (T, S.out)  /* function for synthesized attribute S.live */
 case rootOf(T) of
   ......
   [S -> while E do $S_1$]:
        E.use := R-{E.use} (T[1]) ;  /*  T[1] is the first son of T  */
        $S_1$.out := emptySet ;
        $S_1$.live := emptySet ;
        /* repeat until all attribute values in the cycle become the same as
           the  values computed in the previous iteration */
        repeat
           stop := true ;
           TMP_$S_1$.out := E.use $\cup$ $S_1$.live $\cup$ S.out ;
           if TMP_$S_1$.out $\neq$ $S_1$.out then stop := false ;
           $S_1$.out := TMP_$S_1$.out ;
           TMP_$S_1$.live := R-{S.live} (T[2], $S_1$.out) ;
           if TMP_$S_1$.live $\neq$ $S_1$.live then stop := false ;
           $S_1$.live := TMP_$S_1$.live ;
        until stop ;
        return (E.use $\cup$ $S_1$.live $\cup$ S.out) ;
   end /* of R-{S.live} */;
...
/* Initialize: let the initial values of attributes in the cycle [x] be empty sets */
for each y in [x] do y := bottom ;
   ...
```

Fig. 6 The attribute evaluator for grammar G1 (actually in Lisp for Jun)

Intermediate Tree and Jun Description

The intermediate tree used in our environment is common to all back-end phases. It is in the form of an abstract syntax tree, similar to DIANA [Goos 83].

The AG description that is the input to Jun is similar to the usual AG. A difference is that Jun is based on tree grammars and the syntax is in the form of the tree. An example Jun description is shown in Fig. 7.

```
%class
      STM ::= stm_s|assign|if|while|proc_call|                       ...(a)
              write|writeln|read|null_stm;
...
%node
      stm_s  =>   STM, STM;                                           ...(b)
      assign =>   var : C_ID, EXP;
      if     =>   COND, STM;
      while  =>   COND, STM;
...
%attribute
      C_BLOCK, C_PROCDECL, C_PROC_ID, STM =>                          ...(c)
                  out  : inh,
                  in   : synt save;                                   ...(#)
      C_CONSTDEF, C_VARDECL,
      EXP, COND, BOP, UOP, C_ID, C_NAME, C_NUMBER =>
                  use  : synt;
...
```

Fig. 7 Jun description for live variable analysis of PL/0 (part) (cont. on next page)

```
%semantics
stm_s    { stm2.out = stm_s.out ;                                              ...(d)
           stm1.out = stm2.in ;
           stm_s.in = stm1.in }
null_stm { null_stm.in = null_stm.out }
assign   { assign.in = (union exp1.use
                          (set-difference assign.out var.use)) }
if       { stm1.out = if.out ;
           if.in = (union cond1.use stm1.in) }
while    { %circle stm1.out, stm1.in ;                                         ...(*)
           stm1.out = (union (union cond1.use while.out) stm1.in) ;
           while.in = (union (union cond1.use while.out) stm1.in) }
...
```

Fig. 7 Jun description for live variable analysis of PL/0 (part)

The specification of the syntax (structure) of the tree incorporates *nodes* and *classes*. A *node* actually means the *node type* appearing in the (abstract) syntax tree (see Fig. 7 (b)). A *class* is a collection of node types of the syntax tree (see Fig. 7 (a)). This is for collecting similar node types (e.g. 'assignment statement', 'conditional statement', 'while statement', etc.) into a single category (e.g. 'statement'). In Jun, class names are conventionally written in upper case letters.

The attribute definitions come down next (see Fig. 7(c)). An attribute can be associated to one or more classes. Here 'synt' and 'inh' mean synthesized and inherited attributes, respectively. Then come the semantic rules (see Fig. 7(d)), which are normally in the form of

node name or class name { semantic rules } .

We allow *local attributes* in the Jun description similar to the ones in the Rie description.

In the synth-function evaluator of Jun, we allow two modes of evaluation for each synthesized attribute: the *usual mode* and the *save mode*. In the usual mode, the evaluated value of a synthesized attribute is returned as the function value and is not stored in the tree. In the save mode, the evaluated value is stored in the corresponding node of the tree, and after that the stored value is taken without reevaluating it. The save mode of evaluation can be specified by putting an option '`save`' to the relevant attribute (see Fig. 7(#)).

The generator can detect circularities of attribute dependency, but it cannot decide whether the attributes in the cycle and the relevant semantic rules satisfy the condition of finitely recursive AG shown before. Thus, we decided that the writer of the AG should be responsible to see that the conditions are satisfied, and (s)he should specify such attributes by

`%circle` attribute occurrence, ... ;

at the beginning of the semantic rules (see Fig. 7(*)). This strategy to let the AG writer specify such attributes is less error-prone and more maintainable than the fully automatic strategy. Of course the generator signals an error if attributes in detected cycles are not specified by '`%circle`'.

4. Generation of Optimizer

The optimization phase of a compiler is roughly divided into two components: data-flow analysis and optimizing transformation. The data-flow analysis component, such as the analysis of 'available expressions', 'reaching definitions', and 'live variables', can be generated using Jun. The finitely recursive attribute grammar, on which Jun is based, is well suited for this since most data-flow information is represented as a set and has a least fixed-point.

An example Jun description of the live variable analysis for PL/0 was already shown in Fig. 7. This corresponds to the formal AG specification of Fig. 4.

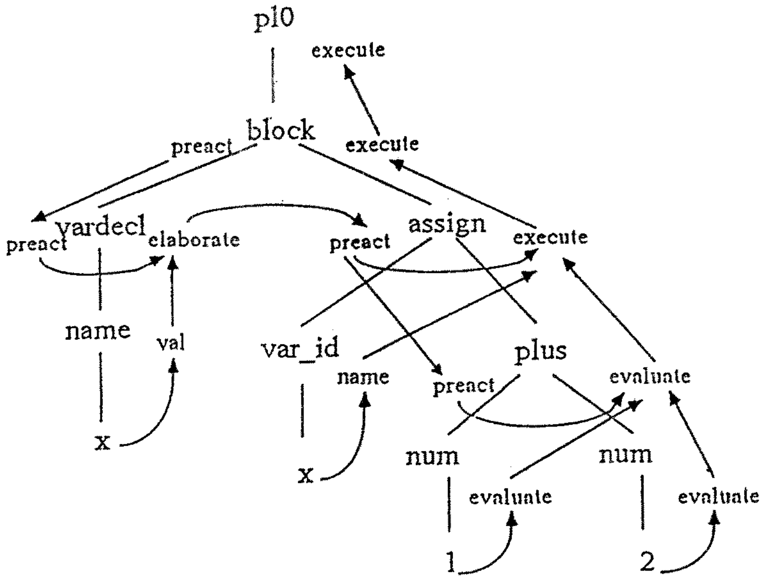
The optimizing transformation component of this phase is not yet well studied.

5. Generation of Interpreter

Several interesting ideas about the formalization of dynamic semantics have been studied. But our main interest was in the generation of interpreters based on the same formalism used in other back-end phases. For that, we borrowed the concept of *action semantics* [Watt 86], which unifies numerous different domains of the denotational semantics into a single domain called *action*. In our work, an action is the run-time environment. We use the following *dynamic attributes*:

```
var  x;
begin
   x:=1+2
end.
```

(a) source program



(b) flow of attributes for (a)

Fig. 8 Flow of attributes in a PL/0 interpreter

elaborate	run-time environment as a result of allocating stores to each variable (result of variable elaboration)
execute	run-time environment after executing a statement (result of executing a statement)
evaluate	value of an expression (result of evaluating an expression)
preact	run-time environment before executing a statement or evaluating an expression

An example of the flow of attributes using the above dynamic attributes is shown in Fig. 8. A part of the corresponding Jun description for PL/0 interpreter is shown in Fig. 9.

```
%semantics
...
assign  /* as_id expl */                                              ...(a)
 { expl.preact = assign.preact ;                                      ...(b)
   assign.execute = (update assign.preact as_id.name expl.evaluate) }...(c)
```

Fig. 9 Description of PL/0 interpreter (part)

For example, at the node for assignment (`assign`) of Fig. 9(a), the evaluator let the run-time environment before evaluating the expression in the right-hand side (`expl.preact`) be the environment before executing the assignment (`assign.preact`) (see Fig. 9(b)). Next, the evaluator updates the variable in the left-hand side (`as_id.name`) to the value which results from evaluating the expression in the right-hand side (`expl.evaluate`), and the resulting environment becomes the environment after executing the whole assignment (`assign.execute`) (see Fig. 9(c)).

Since attribute evaluation in the synth-function evaluator of Jun is *demand driven*, only necessary parts are executed. Therefore, conditional statements, such as an 'if statement', can be handled correctly.

Now, we had a serious problem in loop statements. As an example, consider the dependency of dynamic attributes at a node of a 'while statement'

```
          while  =>  cond   stm_s .
```

The dependency is shown in Fig. 10.

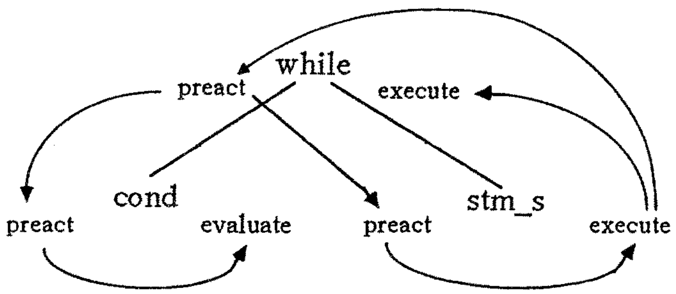


Fig. 10 Dependency of dynamic attributes of 'while statement'

In this 'while statement', the whole statement is usually re-executed several times after executing `stm_s`. So there is an attribute dependency from `execute` of `stm_s` to `preact` of `while`. This constitutes a cycle in attribute dependency at run-time and attributes in this cycle are usually evaluated several times. Note that this problem is essentially caused by *dynamic semantics* that deals with the run-time behavior, and cannot be resolved by the iterative evaluation of finitely recursive attribute grammars which deals with the *static semantics*. The attribute grammar was originally conceived for describing static semantics at compile-time, and in pure AGs, attribute values have the property of single assignments and their values cannot be modified once they are defined. Therefore, AGs cannot completely specify the dynamic semantics, unless we use functions themselves as attribute values.

Our solution here was made on a practical standpoint. We didn't go deep into the single assignment property of AGs: *we allowed re-assignment to attributes if the assignment was explicitly described as a side-effect within the right-hand side of semantic rules*. Fig. 11 shows our way of describing dynamic semantics of a 'while statement'. Here, `while.preact` is re-assigned its value in the right-hand side of a semantic rule (Fig. 11(*)).

```
while  /* as_cond as_stm_s */
     { as_cond.preact = while.preact ;
       as_stm_s.preact = while.preact ;
       while.execute = (loop (when (not as_cond.evaluate)
                                  (return while.preact))
                              (setq while.preact as_stm_s.execute)) }...(*)
```

Fig. 11 Description of dynamic semantics of 'while statement'

The above strategy is of course not a complete solution. But considering that the possibility of cycles in attribute dependency at run-time is generally restricted to loop statements, the part exceeding the pure AG formalism is quite limited. Moreover use of the same attribute evaluator as in other back-end phases was attractive in our generation environment.

6. Generation of Code Generator

There are several formalism for the code generator, such as the one using pattern matching and tree-rewriting rules [Cattell 80], LR parsing techniques [Glanville 78], and affix grammars [Ganapathi 85]. We adopted the attribute grammar formalism based on tree grammars, because multiple traversals over the syntax tree are possible in AGs, which cannot be performed in other methods. Using AGs enables description of algorithms such as the generation of shortest object code for expressions using minimum registers. Moreover, adoption of the unified formalism for all back-end phases was attractive.

An example description of code generator for PL/0 is shown in Fig. 12. This is a part of the description for generating the shortest object code of expressions for Sun-3. Four types of nodes - plus, minus, mul, and div nodes - are treated together. Here the attribute *regs* is the set of allocatable registers, *code* is the generated code, *treg* is the target register to which the result value is stored, and *nregs* is the required number of registers.

The algorithm for the shortest code generation for expressions is usually given in the form of two-pass traversals on the syntax tree [Aho 86]. However, we think such a description is too procedural and specifies the order of processing more than is necessary. If we write this algorithm in AG as in Fig. 12, the description is declarative and easy to read. We don't need to think about the order of tree traversals. As an example of object code, take the expression

a*(b+c)-d*e/(f-g) .

The code for this expression generated by the vendor-supplied compiler (Sun-3 Pascal) uses 3 registers, while the code generated by the AG of Fig. 12 uses 2 registers, which is optimal.

We think that use of AGs for code generators is well suited to the generation of codes for RISC machines. However in CISC machines, methods using pattern matching of trees may yield better results by considering numerous matching patterns for a given tree.

```
BEXP /* as_expl as_exp2 */ /* binary expressions plus, minus, mul and div */
{%local n1 = (case %node
                ((plus mul) (if (= 0 as_expl.nregs as_exp2.nregs) 1 as_expl.nregs))
                ((minus div) (max as_expl.nregs 1))),
        n2 = as_exp2.nregs,
        op = (case %node
                ('plus   "addl")
                ('minus  "subl")
                ('mul    "mulsl")
                ('div    "divsl"));
 as_expl.regs = (request-reg (case %node
                                   ((plus mul) (if (>= n1 n2)
                                                   BEXP.regs
                                                   (reverse BEXP.regs)))
                                   ((minus div) BEXP.regs))
                             n1);
 as_exp2.regs = ...
 BEXP.code = (cond ((case %node
                          ((plus mul) (and (>= n1 *sum-of-reg*)(>= n2 *sum-of-reg*)))
                          ((minus div) (and (> n2 0)(> BEXP.nregs *sum-of-reg*))))
                      (append as_exp2.code
                                (genop "movl" (to-string as_exp2.treg ",sp@-"))
                                as_expl.code
                                (genop op (to-string "sp@+," as_expl.treg))))
                   ((>= n1 n2)
                      (append as_expl.code as_exp2.code
                                (genop op (to-string as_exp2.treg "," as_expl.treg))))
                   (t (append as_exp2.code as_expl.code
                       (genop op
                        (case %node
                         ((plus mul) (to-string as_expl.treg "," as_exp2.treg))
                         ((minus div) (to-string as_exp2.treg "," as_expl.treg)))))));
 as_expl.left = (case %node
                      ((plus mul) (>= n1 n2))
                      ((minus div) t));
 as_exp2.left = (case %node
                      ((plus mul) (< n1 n2))
                      ((minus div) nil));
 BEXP.treg    = (case %node
                      ((plus mul) (if (>= n1 n2) as_expl.treg as_exp2.treg))
                      ((minus div) as_expl.treg));
 BEXP.nregs   = (if (= n1 n2) (1+ n1) (max n1 n2));
}
```

Fig. 12 Description of code generator for PL/0 (part)

7. Experimental Results

The number of lines (including comments and blank lines) for the description of each phase of a compiler for the PL/0 language using Rie and Jun, and the corresponding generation time are as follows.

	---------------------------- description	-------------------			-- generation --
	declaration of symbols, nodes, attributes etc. (lines)	syntax and semantic rules (lines)	others (lines)	total (lines)	time (sec)
front-end (Rie)	82	539	353(in C)	974	0.81
optimizer (Jun) (data-flow analysis)	96	317	79 (*)	492	54.8
interpreter (Jun)	112	191	74 (*)	314	35.2
code generator (Jun)	75	282	65 (*)	366	50.9

(*) in Common Lisp

It can be seen that the generation time by Rie is quite fast. Note that in Jun, the number of description lines was reduced by about 40%, by collecting node types with similar semantic rules into a class. The time of generation is on Sun-4/280.

8. Concluding Remarks

We have presented an attempt of our group to generate almost all compiler phases by using compiler generators based on attribute grammars.

The following issues are left as future problems.

(i) The transformation component of the optimizer is not well studied. It should be further investigated. Transformation of trees [Lipps 89] seems to be useful.

(ii) The description of dynamic semantics for the interpreter was made on a practical standpoint. However, a better formalism might be found.

(iii) So far the formalism is made for structured syntax of programming languages without *goto statements*. Application to goto statements and other non-local jumps should also be investigated.

(iv) We do not have a description of back-end phases for large scale programming languages. Further experience would be required before languages of practical size could be handled.

(v) Jun is presently a prototype. More detailed design and efficient implementation are desirable.

Acknowledgments

The author wishes to thank the members of the programming language laboratory for assistance in the development of Rie and Jun, especially H. Ishizuka, M. Sawatani, J. Iizumi, and to I. Nakata and K. Hayashi for helpful comments to the manuscript.

References

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

[Babich 78] Babich, W.A. and Jazayeri, M. : The Method of Attributes for Data Flow Analysis, Part I. Exhaustive Analysis, Acta Inf. 10, 245-264 (1978).

[Cattell 80] Cattell, R.G.G. : Automatic Derivation of Code Generators from Machine Descriptions, ACM TOPLAS, 2, 2, 173-190 (1980).

[Farrow 84] Farrow, R. : Generating a Production Compiler from an Attribute Grammar, IEEE Software, 1, 4, 77-93 (1984).

[Farrow 86] Farrow, R. : Automatic Generation of Fixed-Point-Finding Evaluator for Circular, but Well-Defined, Attribute Grammars, ACM SIGPLAN '86 Symp. on Compiler Construction, 85-98 (1986).

[Ganapathi 85] Ganapathi, M. and Fischer, C. N. : Affix Grammar Driven Code Generation, ACM TOPLAS, 7, 4, 560-599 (1985).

[Ganzinger 82] Ganzinger, H. and Giegerich, R. : A Truly Generative Semantics-Directed Compiler Generator, in Proc. SIGPLAN Symp. on Compiler Construction, SIGPLAN Notices,17, 6, 172-184 (1982).

[Glanville 78] Glanville, R. S. and Graham, S. L. : A New Method for Compiler Code Generation, 5th ACM POPL, 231-240 (1978).

[Goos 83] Goos,G. et al. : DIANA An Intermediate Language for Ada, revised version, Lec. Notes in Comp. Sci., Vol. 161, Springer, 1983.

[Kastens 82] Kastens, U., Hutt, B. and Zimmermann, E. : GAG: A Practical Compiler Generator, Lec. Notes in Comp. Sci., Vol. 141, Springer, 1982.

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

[Koskimies 88] Koskimies, K., et al. : The Design of a Language Processor Generator, Softw. Pract. Exper.,18, 2, 107-135 (1988).

[Kuse 86] Kuse, K., Sassa, M. and Nakata, I. : Modelling and Analysis of Concurrent Processes Connected by Streams, J. Inf. Process., 9, 3, 148-158 (1986).

[Lesk 75] Lesk, M.E. : Lex - A Lexical Analyzer Generator, Computer Science Tech. Rep. 39, AT&T Bell Lab., 1975.

[Lipps 89] Lipps, P. Möncke, U. and Wilhelm, R. : OPTRAN - A Language/System for the Specification of Program Transformations: System Overview and Experiences, Proc. 2nd CCHSC Workshop, Lec. Notes in Comp. Sci., Vol. 371, Springer, 1989.

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

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

[Sassa 88] Sassa, M. : Incremental Attribute Evaluation and Parsing Based on ECLR-attributed Grammars, Report A-1988-9, Dept. of Computer Science, Univ. of Helsinki, 1988.

[Sassa 90] Sassa, M. Ishizuka, H. Sawatani, M. and Nakata, I. : Introduction to Rie and Rie User's Manual, Tech. Rep. ISE-TR-90-82, Inst. of Inf. Sci. & Elec., Univ. of Tsukuba, 1990.

[Shapiro 83] Shapiro, E.Y. : Algorithmic Program Debugging, MIT Press, 1983.

[Watt 86] Watt, D. A. : Executable Semantic Descriptions, Softw. Pract. Exper., 16, 13-43 (1986).

[Yamashita 88] Yamashita, Y. and Nakata, I. : Programming in Gramp: a programming language based on CCFG, Tech. Rep. ISE-TR-88-73, Inst. of Inf. Sci. & Elec., Univ. Tsukuba, 1988.