SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 22(5), 419–437 (MAY 1992)

A New Parsing Method for Non-LR(1) Grammars

A. GAYLER HARFORD*, VINCENT P. HEURING† AND MICHAEL G. MAIN
Departments of Computer Science and Electrical and Computer Engineering, University of Colorado at Boulder, Boulder, CO 80309-0425, U.S.A.

SUMMARY

One of the difficult problems that faces a compiler writer is to devise a grammar that is suitable for both efficient parsing and semantic attribution. This paper describes a system that resolves conflicts in LR(1) parsing by taking advantage of information in the parse tree. The system, which functions as part of a compiler generator, rewrites the user's grammar to remove parsing conflicts. It then places code into the generated compiler that rewrites the parse tree during parsing so as to produce the tree of the original grammar. The compiler writer can then write the semantic attribution to fit his or her original grammar without any knowledge of the changes made. The method is expected to be efficient in most cases, even in parsing systems that do not explicitly build the entire parse tree. The method complements previous work in its capabilities and advantages. The system has been implemented and integrated into a compiler generator system.

KEY WORDS Parsing Compiler Eli Yacc LR LALR

INTRODUCTION

The advent of compiler generators has put compiler writing within the reach of non-specialists. Although one tends to associate compilers with the translation of programming languages, compiler techniques are applicable to a wide range of problems that have in common the processing of a linear text.

One of the difficult tasks that remains for the user of a compiler generator is to design a suitable grammar. Numerous authors have argued for the need to loosen the constraints of LR(1) and LALR(1) parsing.[1–8] This is particularly important if compiler generators are to serve as tools for rapid prototyping by nonspecialists.

The grammar must, of course, describe the language to be processed so that the compiler can determine whether the input text is a legal utterance of the language. But two other considerations are important. First, it must be possible to build a reasonably efficient parser for this grammar. Secondly, the rules of the grammar must provide a suitable framework for associating semantic meaning with the input. These two considerations impose unrelated and sometimes contradictory constraints with the result that grammar design can be an extended and frustrating process.

* Present address: Intellistor Inc., 2402 Clover Basin Rd., Longmont, CO 80503, U.S.A.
† Author to whom correspondence should be directed.

0038–0644/92/050419–19$09.50

Received 29 October 1991
Revised 30 December 1991

As an example, consider the following grammar fragment:

⟨ *statement* ⟩ → ⟨ *assignment* ⟩
⟨ *assignment* ⟩ → ⟨ *leftside* ⟩ = ⟨ *expression* ⟩;
⟨ *leftside* ⟩ → *identifier*
⟨ *statement* ⟩ → ⟨ *condition* ⟩ : ⟨ *action* ⟩
⟨ *condition* ⟩ → ⟨ *expression* ⟩ = ⟨ *expression* ⟩
⟨ *expression* ⟩ → *identifier*

One symbol of look ahead is insufficient to distinguish between ⟨*leftside*⟩ and ⟨*expression*⟩ (the grammar may not even be LR(k) if expressions may be arbitrarily long strings). Removing the conflict requires not only rewriting the grammar but rethinking the semantic attribution as well.

We have devised an enhancement of LR parsing for compiler generators which loosens the parsing constraints while leaving the user's semantic attribution intact.

A grammar may be viewed as describing a set of parse trees in which each internal node represents a production of the grammar and each leaf corresponds to a basic symbol in the language. The leaves of the tree, taken left-to-right, correspond to a sentence in the language. The parse tree contains information that goes beyond the information inherent in each node. This is information about the relations between productions. Some parsers explicitly build the parse tree in memory so that this information can be used in the semantic computations. We propose that this information can also be used during the parsing process itself to resolve parsing conflicts.

This paper describes a tree rewriting algorithm called *tris* (tree rewriting system) which solves parsing conflicts by rewriting the grammar. *Tris* then puts code into the compiler that rewrites appropriate fragments of the parse tree during parsing so that the final tree corresponds to the original unmodified grammar. In a parser that does not explicitly build the parse tree during parsing, the code generated by *tris* would construct just those parts of the tree needed to ascertain the correct parse.

This scheme relieves the compiler writer of much of the burden of constructing a grammar suitable for the mechanics of parsing and leaves the writer free to tailor the grammar to the needs of the semantic attribution. The semantic attribution can be written for the original grammar without reference to the changes which *tris* has made. The system can be integrated into a compiler writing system so that the compiler writer need not even be aware of the changes that have been made to the grammar.

An experimental implementation of *tris* has been integrated into the *Eli* compiler generating system.[9] We will show that *tris* can be used to generate a working compiler of reasonable efficiency. In addition, we will show that the approach is suitable for other compiler generator systems and that the rewriting of fragments of the parse tree should not unduly affect the parsing efficiency in systems that do not explicitly build the parse tree.

The following sections informally describe and discuss the method. Formal proofs will be found in Reference 10.

INFORMAL DESCRIPTION OF THE ALGORITHM

Case I conflicts

Consider the following simple grammar:

$1: S \rightarrow Q$
$2: Q \rightarrow A\ x\ y$
$3: Q \rightarrow B\ x\ z$
$4: A \rightarrow C\ a\ b$
$5: B \rightarrow C\ a\ b$
$6: C \rightarrow c$

Here and in what follows, upper case Roman letters represent non-terminal symbols, and lower case letters terminal symbols. Each production will be assigned a unique number. We will consider bottom up LR(1) parsing[11] (see References 12–14 for a general discussion of LR(1) parsing).

This grammar is not LR(1) since productions 4 and 5 cannot be distinguished with only one symbol of look ahead. However, it is clear that the grammar is not ambiguous since the reading of a y or z at the end unambiguously distinguishes between productions 2 and 3 and hence between the child productions 4 and 5.

This conflict can be removed by replacing productions 4 and 5 with a single production

$7: K \rightarrow C\ a\ b$

Two new productions corresponding to 2 and 3 are now added.

$8: Q \rightarrow K\ x\ y$
$9: Q \rightarrow K\ x\ z$

Since there are no remaining productions with A or B on the left, neither of productions 2 and 3 can be part of a derivation. Thus we can remove them from the grammar. In the next example, we shall examine a case where they must be left in the grammar.

Now we have removed the parsing conflict. However, we have not made a distinction between A and B. This uncertainty can be resolved when either production 8 or 9 is reduced. So we might build into our parsing machine the knowledge that when production 8 is reduced, K corresponds to A and the previously reduced production 7 corresponds to production 4. Likewise, we could build in the knowledge that when production 9 is reduced, K corresponds to B and the previously reduced production 7 corresponds to production 5.

We will refer to productions 8 and 9 as resolving_parents since their reduction provides information about a child that resolves the original parsing conflict. Productions 4 and 5 respectively will be called the resolutions brought about by the reductions of the resolving parents. The position of the resolved child on the right side of the resolving_parent will be called the birth_order. Positions are numbered from left to right starting with 1. Here the birth_order of the resolved child is 1. We

will call productions 2 and 3 the old_parents of productions 8 and 9 respectively. old_parent is the production in the original grammar to which the resolving_parent corresponds. The knowledge needed to rewrite the parse tree can be represented by a function resolve, which takes two arguments and returns two values:

resolve(resolving_parent, birth_order) = (resolution, old_parent)

We may write

resolve(8,1) = (4,2)

Likewise

resolve(9,1) = (5,3)

Figure 1 shows one of the transformations that our enlightened parser would make, which can be thought of as rewriting the parse tree.

The second example is slightly more complicated. Let us add to the original grammar another production

$$10: A \rightarrow C\ d$$

Now if we were to make our correction as before, there would be no way for production 10 to be part of a derivation because production 2 would have been

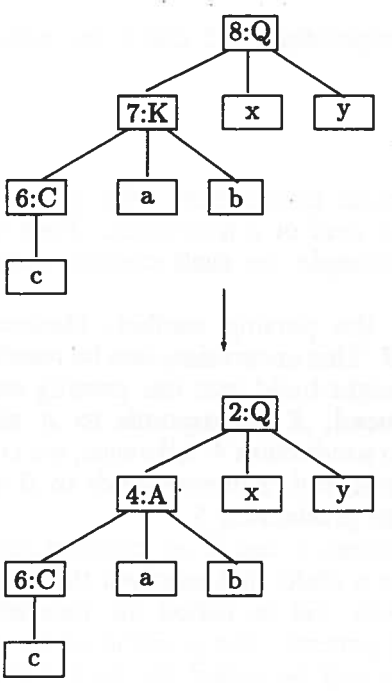

Figure 1. Rewriting the parse tree for resolution: resolve(8,1) = (4,2)

removed. We could handle this situation easily by not removing production 2 when we introduce production 8. Now let us add to the original grammar an additional production:

$$11: Q \rightarrow C\ A$$

Now when we replace productions 4 and 5 with 7, we will need a further change to the grammar to preserve the language. One possibility is to add a new production

$$12: Q \rightarrow C\ K$$

However, it is more convenient to handle this case in another way: Replace 11 with

$$13: Q \rightarrow C\ J$$

and add two productions

$$14: J \rightarrow A$$
$$15: J \rightarrow K$$

These changes allow us to derive all the strings we could before. Now when 15 is reduced we know that the child of J represents production 4 in the original grammar. We can represent this knowledge with function resolve as before with one modification. There is no old_parent since production 15 has no counterpart in the original grammar. We will use 0 to represent the absence of an old_parent. Thus in this example

resolve(15,1) = (4,0)

But note that now our parse tree has a different structure from the original one because of productions 14 and 15 (see Fig. 2, lower left). To remedy this, we will build into our parser the knowledge that when production 13 is reduced, the J child (child 2) should be eliminated and the grandchild put in its place.

Figure 2 shows these transformation of the parse tree. The upper half of the figure shows the transformation that would be carried out upon reduction of production 15

$$15: J \rightarrow K$$

The lower half shows the subsequent transformation to be carried out when this J becomes a child of Q in production 13.

We will refer to production 13 as the swallow_parent because it must 'swallow' its child to produce the original parse tree. The birth_order is the position of the child to be swallowed, 2 in this case. We will refer to production 11 as the old_parent because it is the production in the original grammar to which the swallow_parent corresponds. We can represent this knowledge with a function swallow which takes two arguments and returns one value:

swallow(swallow_parent,birth_order) = old_parent

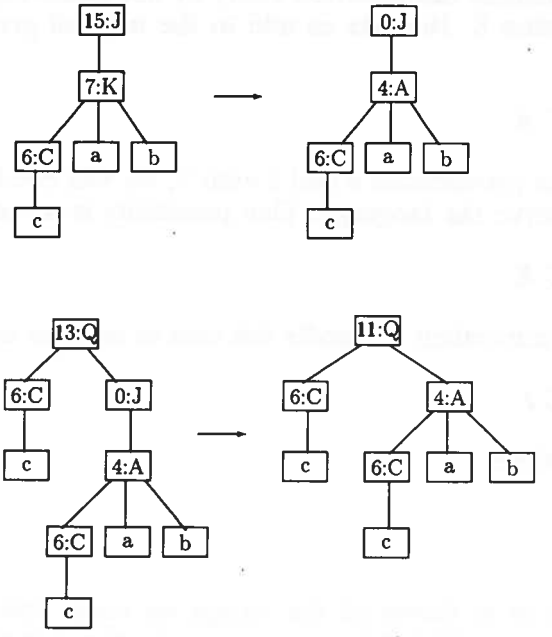

Figure 2. Upper: rewriting the parse tree to resolve a J *production:* resolve(15,1) = (4,0)*; Lower: rewriting the parse tree for swallowing:* swallow(13,2) = 11

Note that in the case of bottom up parsing, at the time the J child is swallowed by its parent, the true identity of the grandchild in the original grammar has already been established.

Figure 2 shows these transformations of the parse tree. The upper half of the figure shows the transformation that would be carried out upon reduction of production 15

$$15: J \rightarrow K$$

The lower half of the figure shows the subsequent transformation to be carried out when this J becomes a child of Q in production 13.

The swallow method is convenient to use since in general there may be many productions to deal with, and making alternatives to include every combination of K and A leads to a large number of extra productions. Observe, however, that the swallow method will be useful only if J is placed at a position where no conflict has arisen. Note that the swallow method is of no use in our original example since it will not eliminate the original conflict.

We will refer to this type of reduce–reduce conflict, where the right sides of the conflicting productions are the same, as a *case I conflict*. The transformation we have described above, which rewrites the grammar and defines the functions resolve and swallow is defined formally in Reference 10.

We will say that a transformation fails if it produces two identical productions, that is, two productions having identical left and right sides. In this situation the transformation is of no use to us since a grammar with identical productions is

ambiguous. The transformation will also be said to fail if either of the original conflicting productions has the start symbol on the left.

We have turned this seeming limitation to our advantage by showing that if the transformation fails then at least we may conclude that the original untransformed grammar is ambiguous (Reference 10, Theorem 2). In our experience, parsing conflicts often result from unrecognized ambiguities in the grammar. Application of the above transformation will identify some of these ambiguities.

When a transformation succeeds we will call the new grammar produced, together with the two functions resolve and swallow, a *case I correction*. We have shown that both the case I correction grammar and the original uncorrected grammar describe the same language (Reference 10, Theorem 3). Furthermore, there is a one-to-one correspondence between the parse trees of the two grammars (Reference 10, Theorem 3). Thus the successful transformation does not entail any loss of information. Information lost at one level of a parse tree is present at another higher level. It is this property of the case I correction that make it useful to our new parsing method.

Case II conflicts

Consider the following grammar:

$1: S \rightarrow Q$
$2: Q \rightarrow A\ x\ y$
$3: Q \rightarrow B\ c\ y$
$4: A \rightarrow a\ b\ c$
$5: B \rightarrow a\ b$

Here we have a shift-reduce conflict after reading $a\ b$ since we do not know whether to reduce production 5 or to wait for the possibility of production 4. This is an example of what we call a *case II conflict*.

We can resolve the conflict if we can manage to wait until an x or y is read after the c which is expected next. We can postpone the decision by rewriting the grammar in the following way. We will create a new production

$6: K1 \rightarrow a\ b$

Now production 4 can be replaced with a new production

$7: A \rightarrow K1\ c$

We have now avoided our original conflict, but have a new conflict between productions 5 and 6. But this conflict is case I, which we discussed above. It can be eliminated by replacing productions 5 and 6 with

$8: K2 \rightarrow a\ b$

and productions 7 and 3 with

$9: A \rightarrow K2\ c$

$$10: Q \rightarrow K2\ c\ y$$

respectively.

Figure 3 shows how the parse tree can be rewritten when production 9 is reduced. The top transformation is described by

resolve(9,1) = (6,7)

The additional knowledge needed for the bottom transformation can be represented by a function

expand(expand_parent,birth_order) = old_parent

For this situation we have

expand(7,1) = 4

When production 10 is reduced, the parse tree is rewritten according to

resolve(10,1) = (5,3)

There are several variants on the above conflict where the expand technique will not work. In these cases the conflict can be eliminated by embedding one of the productions in its parent. For example, consider the following simple grammar:

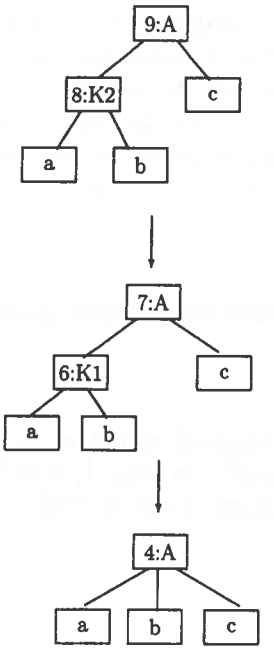

Figure 3. Rewriting the parse tree for resolution followed by expansion: resolve(9,1) = (6,7); expand(7,1) = 4

$1: S \rightarrow Q$
$2: Q \rightarrow A\ x\ y$
$3: Q \rightarrow a\ B\ z$
$4: A \rightarrow a\ b\ c$
$5: B \rightarrow b\ c\ x\ y$

After reading the string *abc* one symbol of lookahead (or even two) will not tell us whether to reduce production 4 or continue to consider production 5. This conflict can be eliminated by embedding production 4 into production 2 to produce

$6: Q \rightarrow a\ b\ c\ x\ y$

When production 6 is reduced, we will 'pop out' a child

$4: A \rightarrow a\ b\ c$

to produce the tree of the original grammar as shown in Figure 4. This process is described by a function

popout(popout_parent,birth_order,poplength) = (popchild,old_parent)

Here poplength is the number of children to be removed from popout_parent and birth_order is the position of the leftmost child to be removed. These children become the children of the new node popchild. As a result of the transformation, popout_parent becomes old_parent.

Two other situations where the popout method is appropriate are detailed in the formal treatment of case II in Reference 10. Note that there is no benefit from embedding a production into itself.

As in the case of a case I correction there is no loss of information accompanying

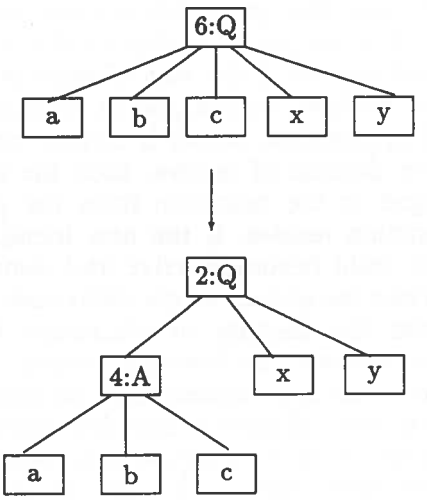

Figure 4. Rewriting the parse tree for popout: popout(6,1,3) = (4,2)

a case II correction. Both the case II correction and the original uncorrected grammar describe the same language, and there is a one-to-one correspondence between the parse trees of the two grammars (Reference 10, Theorem 4).

Case I and case II together cover essentially all situations in the sense that except for conflicts involving empty productions in the initial item set, for every conflict that is not a case I conflict there exists a case II conflict (Reference 10, Lemma 1).

THE ALGORITHM

The algorithm makes a sequence of case I and case II corrections to the starting grammar, hopefully terminating in an LR(1) grammar. Failure of a case I correction is reported to the user as an ambiguity in the original grammar.

The design of the algorithm takes advantage of the fact that all production numbers are unique. Each of the functions resolve, swallow, expand, and popout is a set of ordered pairs that comprises the mapping from its domain to its range. Each time a correction is made that involves one of these functions, a new pair is added to the function. We can guarantee that the analysis will terminate only by putting an upper limit on the number of cycles that will be carried out before giving up.

Parsing is carried out in the conventional bottom up manner using the final grammar obtained after all of the corrections. The simplest way to describe the algorithm is in terms of a parser that builds the entire parse tree. This is the description detailed below. However, it is easily seen that the method can be adapted to a parser that ordinarily does not build the parse tree. In this latter case, only the nodes of the tree needed by the algorithm are constructed.

Nodes of the tree that are in the domains of functions swallow, expand, popout, or resolve are termed active nodes since they can initiate a transformation immediately after they are formed. A node that must wait for action from its parent to be swallowed, expanded, or resolved, is said to be passive.

Immediately upon formation of an active node the appropriate action is carried out:

1. If the node is in the domain of swallow or expand, then the appropriate child node is eliminated, and the grandchildren are elevated to the position of children. The node then assumes the identity of old_parent, that is returned by functions swallow and expand. If the new identity of the parent node is now in the domain of any of swallow, expand, popout, or resolve, then the node remains active and the next appropriate action is carried out.
2. If the node is in the domain of resolve, then the identity of the appropriate child node is changed to the resolution from the pair (resolution, old_parent) returned by the function resolve. If the new identity of the child node is an active one, then the child becomes active and remains so until it receives an inactive identity. When the actions of the child node have run their course, the parent then receives the identity of old_parent from the pair (resolution, old_parent) that was returned by function resolve. If this new identity is an active one, then the node is processed until an inactive identity is received.
3. If the node is in the domain of popout, then the appropriate children are popped out and made grandchildren by creation and insertion of a new child node. This new child node then receives the new identity pop_child from the pair (pop_child, old_parent) returned by function popout. If the new child node

receives an active identity then it is processed as an active node. When its processing has run its course, the parent node receives the identity old_parent. If this new identity is now an active one, the node is processed accordingly.

The recursive procedure process_active_node shown below summarizes the above. Here the procedure finish_node confers upon a node its identity in the final parse tree upon which attribute computation will be carried out. Note that the parse tree is being rewritten during the parsing process.

The correction information is stored in a table, fix_list, which is an array of records of fix_type.

```
type
  kind_type=(wait,finish,expand,swallow,resolve,popout);
  fix_type = record
    next_fix_rule, which_child, resolution: integer;
    kind: kind_type
    original_rule: rule_type;
    original_nonterminal_symbol: nonterminal_type;
  end;
procedure process_active_node(active_rule : integer;active_ptr : nodeptr);
(* nodeptr is a pointer to the record used to represent a node of the
   the parse tree; active_ptr is a pointer to the active node
   to be processed*)
var
  active: boolean;
  fix_rule, i: integer;
  sonptr: nodeptr;
begin
  fix_rule := active_rule;
  active := true;
  while active do
  begin
    if fix_list[fix_rule].kind=resolve then
    begin
      sonptr := pointer to son to be resolved

      (* disambiguate the son *)
      process_active_node(fix_list[fix_rule].resolution,sonptr);
    end (*if kind=resolve*)
    else if fix_list[fix_rule].kind=popout then
      (*pop out next num_popout of children and return pointer
        to new child (pop_out_children); then process new child*)
      process_active_node(fix_list[fix_rule].disam_rule,
        pop_out_children(act_ptr, fix_list[fix_rule].which_child,
        fix_list[fix_rule].num_popout))
    else
      case fix_list[fix_rule].kind of
```

```
            wait:;
            finish: finish_node(fix_rule,active_ptr);
            expand: expand_child(fix_rule,active_ptr);
            swallow: swallow_child(fix_rule,active_ptr);
          end;
        if fix_list[fix_rule].kind = finish then
          active := false
        else
          fix_rule := fix_list[fix_rule].next_fix_rule;
      end; (*while*)
    end; (*process_active_node*)
```

The algorithm works because each node undergoes in reverse order the sequence of changes it underwent during the sequence of case I and case II corrections. Each node that must be changed starts with the last production number, which it receives when it is formed upon reduction. If this is an active number, then upon reduction it will immediately change and continue to change until it has a number in the original grammar or it receives a passive number and must wait for action from its future parent. If that action is resolve or popout, then upon resolution the node then becomes active again.

Observe that since the parsing is bottom up, in cases where a node triggers changes to its children the child node will always be available. The child node will thus eventually attain its final identity if its parent will. This reasoning can be extended inductively up the tree until a node is found that does not have a passive state along its path to its final identity. We are guaranteed that such a node will be found since the root of the tree cannot have a passive state.

Efficiency of the algorithm

If one considers only the final grammar that the algorithm produces, without any tree rewriting during parsing, then the time complexity of the parsing is the same as that for conventional LR parsing, namely linear with the length of the input text. With the tree rewriting actions, the time complexity increases by only a constant factor that depends upon the grammar. This is true because the number of changes that a given node will undergo is a fixed constant.

The space complexity of LR parsing arises from two sources:

(a) the amount, if any, of the parse tree that is built during parsing
(b) the size of the parse tables.

With respect to the first source, for a parser that explicitly builds the tree, our method increases the complexity of parsing by only a constant factor dependent upon the grammar. However, since the size of the parse tree depends upon the length of the input text, a parser that does not build the tree is in principle much more space efficient.

It is important to realize, however, that there is no theoretical requirement that the entire parse tree be built. Only those tree fragments necessary for the resolution of the parsing conflicts need be built. Furthermore, once a conflict has been resolved, the space for the tree fragment is no longer needed and can be reused. In such a

parser the finish action (see procedure process_active_node in previous section) would initiate the semantic computation associated with the production whose identity had just been established. A second consideration for a parser that does not explicitly build the parse tree is the time of attribute computation. A *GAG*[16] generated compiler does not begin any attribute computations until the entire parse tree has been built. A *YACC* generated compiler, on the other hand, may begin attribute computation immediately upon reduction of the triggering production. Use of our method in a parser such as *YACC* would allow attribute computation to begin as soon as the disambiguating information in the input string arrived.

Despite the inefficiency, there are definite advantages to building the parse tree. The tree provides a structural framework that facilitates specifying relationships between attribute computations in different parts of the tree. For example, *Aladin*, the specification language for *GAG*, has a construction called INCLUDING, which allows the computation at a node to look up the tree an arbitrarily long distance in order to find a needed attribute. There is an interesting parallel between INCLUDING and the correction of a case I conflict by our algorithm.

With respect to the second source of space complexity, the size of the parse tables, it is difficult to generalize from a theoretical point of view. The most meaningful comparison would be between the number of states of the parser produced with our method and the number produced when the original grammar is rewritten by a human to make it LR(1). We would expect an experienced grammar writer to outperform our method in this regard.

All of these considerations must be balanced against the cost of measures that must be taken to make the grammar LR(1) in the absence of our method. This may entail more complicated semantic attribution of the parse tree. A disadvantage in shifting the burden from the context-free grammar to the context-sensitive attribute computations is that generation of the latter part of a compiler has been more difficult to automate than generation of the context-free parser.

IMPLEMENTATION

We have written an experimental system called *Tris I* that carries out the grammar rewriting and tree rewriting discussed above. *Tris I* is conceived as an accessory to an LALR(1) parser generator. If the parser generator finds parsing conflicts in the user's grammar, *Tris I* attempts to rewrite the grammar to eliminate them. If the process is successful, the parser generator reports success to the user, but does not report the changes made to the grammar. The user then writes the semantic attribution for the original grammar.

If the process is unsuccessful, the remaining conflicts are reported to the user in terms of the user's original grammar. Only the right hand sides of the conflicting grammar are written. Whenever a non-terminal occurs that was not in the user's original grammar, the right side of one of its productions is inserted. The insertion is bracketed by ⟨*..*⟩ to indicate its presence.

Tris I has been designed to fit into the *Eli* compiler writing system.[9] *Eli* is a collection of tools for writing a compiler that are managed by the *Odin* system.[17] *Eli* coordinates the functioning of the tools, calling each one automatically when it is needed. In addition, it keeps track of which files must be regenerated after changes have been made and assists the user in pinpointing the source of errors. This

functionality is made possible by *Odin* which is a general system for managing a large software system with a multitude of components. One provides *Odin* with a derivation graph showing how to produce each product of the system.

The principal components of *Eli* are

1. *GLA*, a lexical analyser[18]
2. *PGS*, a parser generator[19]
3. *GAG* or *LIGA*, an attribute evaluator.[16]

A user submits an attribute grammar written in a specification language, *Aladin*. *GAG* reads this specification and determines the order in which nodes of the tree must be visited in order to carry out the computations specified. This function sets *Eli* apart from systems such as *YACC*, where the user must code this functionality himself. *GAG* produces source code that implements the user's *Aladin* specification.

An additional output of *GAG* (with the help of *CAGT*[20]) is a file called pgram which contains the grammar to be submitted to the parser generator along with semantic actions attached to productions. In this case, every production (with some exceptions) has attached a call to either SourceNode or SourceLeaf. These procedures create the nodes of the parse tree during parsing. SourceNode creates a node, pops the child nodes off the stack, attaches the children, and finally pushes the new node onto the stack. SourceLeaf forms a leaf node. It is only after the complete tree has been built that attribute computation will begin. Normally, the user never sees the pgram file. It is automatically submitted to the parser generator at the appropriate time.

PGS uses pgram to produce a parser that executes each SourceNode call as each production is reduced. The output of *PGS* is source code that implements a parser. *Eli* combines this code with code for the lexical analyser, code from *GAG*, possibly code from other tools, and possibly ancillary code from the user to form the source code for a completed compiler.

There is no special notation to be learned to use *Tris I* (except that at present the *PGS* modification option is not supported). The user submits his or her compiler specification to *Eli* in exactly the same way as before. The user is not even aware of what changes *Tris I* has made to his grammar.

The tree constructing actions of *Tris I* are designed to be integrated into the tree construction carried on during parsing. *Tris I* reads a modified form of the pgram file, which is the file that *Eli* normally passes to the parser generator.

Tris I makes a series of passes, each time carrying out an LR(1) analysis of the grammar and correcting one conflict. It stops when either the grammar is LR(1) or a fixed number of passes have been carried out, whichever comes first. It will test the final grammar for LALR(1) conflicts and report them to the user, but it will not fix them.

EXPERIENCE

Tris I has been tested on numerous small, artificial grammars. Reference 10 shows two small example grammars that are successfully handled by *Tris I*. The first grammar describes a fragment of a Pascal-like language. The second describes a notation for a context-sensitive grammar. Also shown for each is the file that *Tris I* submits to the parser generator.

These grammars were given to the students in the compiler construction tools course here at the University of Colorado at Boulder (ECE 5523). Each student was assigned one of the two grammars to rewrite to make it LALR(1). The students experienced considerable difficulty in this task although all eventually succeeded. The LALR(1) versions of the grammar for the context-sensitive notation usually involved shifting some of the burden of syntactic recognition to the semantic attribution.

In addition, *Tris I* has also been tested successfully on some larger grammars up to 100 productions in length. These include grammars for subsets of Pascal and Ada. In the latter case *Tris I* was able to repair the conflict involving procedure identifier is ..., that is discussed by Baker.[4]

Because *Tris I* was designed to be a research tool to gain insight into the algorithm, the full collection of LR(1) item sets is generated for each pass. Consequently, the present implementation of *Tris I* is not practical for large grammars.

We intend to build a more practical implementation of the *tris* algorithm. This version will use an LALR(1) parsing analysis, since the *tris* algorithm is applicable here as well.

DISCUSSION

Although it is often categorically stated that more than one symbol of lookahead is impractical, a number of authors have argued to the contrary (for example, References 2–6 and 8). It is pointed out that when the number of conflicts is small, multi-symbol lookahead is feasible.

Culik and Cohen[1] have suggested a method that would allow parsing of LRR grammars that are not LR(1). There are some serious drawbacks to their method, however. For one thing, the method involves reading the entire text twice. The first time the text is read right to left to accumulate information to help in the parsing decisions during the second read through from left to right. This scheme seems like rather a lot of trouble in order to resolve what is probably a small number of conflicts in most cases. It seems particularly wasteful in cases where the necessary disambiguating information is only a few tokens ahead. Furthermore, a right to left scan may be awkward to carry out, a point emphasized by Baker.[4] A second problem with their method is that no method for automating the necessary grammar analysis was presented. The grammar writer must figure out the scheme for each individual grammar. Despite these drawbacks, the LRR concept introduced by these authors has become an important paradigm for later workers, as described below.

Baker[4] has devised a method for reading ahead using a finite state automaton. Everywhere in the state graph that a transition would occur as a result of a reduction a null transition called a reduce arc is introduced. The resulting state graph can then be used as a finite state machine to extend lookahead. However, he feels that his method is more applicable to error recovery than to resolving parsing conflicts, since it is generally nondeterministic.

Bermudez and Schimpf[5,6,21] have devised a method for parsing an LRR grammar that involves constructing a finite state automaton to process the input text following the conflict in order to make a decision. The construction involves simulating in a breadth first fashion all possible paths of an LR(0) parser from the point of conflict. The method can be fully automated. An arbitrary limit must be placed upon the

lengths of paths in the state diagram that will be considered. The user may specify this limit.

The finite state automaton in the method of Bermudez and Schimpf gives information about the text to the right of the conflict, but none about the text to the left. This would be no problem for an LR(1) parser, but in an LALR(1) parser necessary left context information can be lost. Seite[8] has proposed and automated a method that can resolve the remaining conflicts that result from using an LALR parser by examining the contents of the stack.

The scheme of Bermudez and Schimpf appears to be quite practical in many cases. However, the larger the finite automaton required, the less practical. Our method has the advantage that the space requirements for many grammars that are not LR(k) for any k are not significantly greater than those for LR(1) grammars. Furthermore, with our method no portion of the text is read more than once. With the above methods portions of the text must be read twice, once with the finite state machine and then once with the LR parser.

Other methods for incorporating an extended right context have also been proposed.[22,23]

There are other approaches that do not involve extending lookahead. A number of authors have considered the general problem of rewriting a grammar to make it parsable. The approach taken can be summarized in the concept of a cover. There are a number of formal definitions of cover in the literature.[24,25] We will give an informal description that conveys the flavour of the approach. Suppose we have a grammar G, which is in a form unsuitable for parsing. We would like to find a grammar G' such that any parse using G' can be converted into a parse with G by 'table lookup'. Informally, 'table lookup' means that there exists a function that maps each production in G' to a production in G such that if one applies this function in turn to each production in a parse of G' one gets a parse of the same terminal string in G. (Some authors go farther and demand that there be a homomorphism from the symbols of G' to those of G). A number of interesting results have been obtained showing that broad classes of grammars can be covered by other classes of grammars[24,25] (Reference 25 is a review). However, the reader is cautioned that the covering grammars may be wildly impractical in many cases.[24]

Mickunas, Lancaster, and Schneider[7] have shown that an LR(k) grammar can be covered by an LR(1) grammar with the proviso that derivations of Λ are dealt with separately. Mickunas[26] has shown that even this is theoretically unnecessary, but the necessary transformation may produce exorbitant numbers of productions, many of which are useless. The general idea of their approach is to postpone a decision to reduce a production by adding to the right side of that production the first terminal symbol of the right context, so as to reduce the amount of lookahead by one symbol. The algorithm starts with an LR(k) grammar. Each iteration transforms an LR(i) grammar to an LR($i-1$) one. If the original language is prefix-free, then the covering grammar can be made LR(0). Some simplifications in the algorithm can be made for the sake of efficiency, but they cannot provide that this latter version always halts. The algorithm has been implemented as part of a compiler generator.[27]

It should be noted that the method requires as input an LR(k) grammar, and the larger k the more iterations that must be carried out. Our method does not require an LR(k) grammar, and the number of iterations is independent of the amount of lookahead required by the original grammar.

Of the various previous approaches to resolving parsing conflicts, our method is closest in spirit to that of grammatical covers. However, it is interesting to note that the parsable grammar our method produces is not, in general, a cover for the original grammar. Our work suggests that the notion of a grammatical cover may be too narrow to include all of the grammar rewriting schemes that may be useful for producing a grammar suitable for parsing.

Although we cannot say at the present time how general our method is, it is clear that it complements other methods in its capabilities. Of particular interest, we think, is that our method does not necessarily become more awkward as the amount of lookahead required by the original grammar increases.

CONCLUSIONS AND FUTURE DIRECTIONS

This paper presents a new parsing mechanism for use in conjunction with a parser generator using the LR parsing paradigm. The method is based upon rewriting the parse tree during parsing to resolve parsing conflicts. It has the following features:

1. The method has been proved to produce a correct parser that produces the correct parse tree of the original grammar upon exit. The user may write his semantic attribution to fit the original grammar without any knowledge of the changes that have been made to the grammar.
2. It has been proved that the method will never produce a parser from an ambiguous grammar.
3. The method will detect and report to the user some ambiguities in the original grammar.
4. The time complexity of the parsing process is the same as that for conventional LR or LALR parsing.
5. The space complexity of the parsing process is not greater than that for a parser that explicitly builds the parse tree. With respect to a parser that does not build the tree, the complexity is expected to be not significantly different in most cases.
6. The method can be and has been fully automated.
7. The method has been integrated into the Eli system, a comprehensive compiler development system, and needs virtually no special attention on the part of the compiler writer.
8. The method complements previous work in its capabilities and advantages.

The most pressing practical need of the *Tris I* system as it is presently constituted is to substitute an LALR(1) analysis in place of the rather cumbersome LR(1) analysis now carried out by forming all of the LR(1) item sets. The only drawback to correcting LALR conflicts that are not LR(1) conflicts is that a case I correction may fail without implying the ambiguity of the original grammar. This is probably not serious since the original grammar is not parsable anyway. On the other hand, since ambiguity is a common failing of grammars written for compilers, any information on this is likely to be helpful to the user. This information could be provided by attaching a separate LR(1) analysis unit that could be turned on by the user if desired.

ACKNOWLEDGEMENTS

Portions of this work were supported by the U.S. Army Research Office under contract number DAAL03-86-K-0100, and by the U.S. National Science Foundation under grant number DMC-8506892. This work is based upon the M.S. thesis of AGH. We wish to thank Dr. William M. Waite and the compiler tools group for helpful discussions. Mark Swain, Masayoshi Ishikawa, Robert Gray, and David Bowling provided useful grammars. We also wish to thank the students who tried out the system.

REFERENCES

1. K. Culik II and R. Cohen, 'LR-regular grammars—an extension of LR(k) grammars', *J. Comput. Syst. Sci.*, **7**, 66–96 (1973).
2. M. Ancona, G. Dodero and V. Gianuzzi, 'Building collections of LR(k) items with partial expansion of lookahead strings', *ACM SIGPLAN Notices*, **17**, (5), 25–28 (1982).
3. M. Ancona, G. Dodero, V. Gianuzzi and M. Morgavi, 'Efficient construction of LR(k) states and tables', *ACM Transactions on Programming Languages and Systems*, **13**, (1), 150–178 (1991).
4. T. P. Baker, 'Extending lookahead for LR parsers', *J. Comput. Syst. Sci.*, **22**, (2), 243–259 (1981).
5. M. E. Bermudez and K. M. Schimpf, 'A practical arbitrary look-ahead LR parsing technique', *Proceedings of ACM SIGPLAN Symposium on Compiler Construction* June 1986, pp.136–144.
6. M. E. Bermudez and K. M. Schimpf, 'Practical arbitrary lookahead LR parsing', *J. Comput. Syst. Sci.*, **41**, (1), 230–250 (1990).
7. M. D. Mickunas, R. L. Lancaster and V. B. Schneider, 'Transforming LR(k) grammars to LR(1), SLR(1), and (1,1) bounded right-context grammars', *J. ACM*, **23**, 511–533 (1976).
8. B. Seite, 'A YACC extension for LRR grammar parsing', *Theoret. Comput. Sci.*, **52**, 91–143 (1987).
9. R. W. Gray, V. P. Heuring, S. P. Krane, A. M. Sloane and W. M. Waite, 'Eli: a complete, flexible compiler construction system', *Communications of the ACM*, **35**, (2), 121–131 (1992).
10. A. G. Harford, 'A new parsing method for non-LALR(1) grammars', *Technical Report*, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309, U.S.A., Software Engineering Group Report No. 90-01. 1990. Copies available upon request.
11. D. E. Knuth, 'On the translation of languages from left to right', *Inform. Control.* **8**, 607–639 (1965).
12. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers*, Addison-Wesley, Reading, MA, 1986.
13. J-P. Tremblay and P. G. Sorenson, *Theory and Practice of Compiler Writing*. McGraw Hill, New York, NY, 1985.
14. W. M. Waite and G. Goos, *Compiler Construction*, Springer Verlag, New York, NY, 1984.
15. D. Spector, 'Efficient full LR(1) parser generation', *ACM SIGPLAN Notices*, **23**, (12), 143–150 (1988).
16. U. Kastens, B. Hutt and E. Zimmermann, *GAG: A Practical Compiler Generator*, Springer Verlag, Heidelberg, 1982.
17. G. M. Clemm, 'The Odin system—an object manager for software environments', *Ph.D. Thesis*, Department of Computer Science, University of Colorado, Boulder, CO, 1986.
18. V. P. Heuring, 'The automatic generation of fast lexical analysers', *Software—Practice and Experience*, **16**, 801–808 (1986).
19. J. Grosch and E. Klein, 'User manual for the PGS-system', Gesellschaft fur Mathematik und Datenverarbeitung, Research Institute at the University of Karlsruhe, 1988.
20. A. Bahrami, 'CAGT—an automated approach to abstract and parsing grammars', *M. S. Thesis*, Department of Electrical and Computer Engineering, University of Colorado at Boulder, 1986.
21. M. Bermudez and K. M. Schimpf 'A general model for fixed look-ahead LR parsers', *Int. J. Computer Math.*, **24**, (3+4), 237–271 (1988).
22. R. Farshi, 'LRRL(k) grammars: a left to right parsing technique with reduced lookaheads', *Ph.D. Thesis*, University of Alberta, Edmonton, Alberta, Canada, 1986.
23. J. P. Schmeiser, 'An RLALR(k) parser generator', *M.Sc. Thesis*, University of Saskatchewan, Saskatoon, Saskatchewan, Canada, 1988.

24. J. N. Gray and M. A. Harrison, 'On the covering and reduction problems for context-free grammars', *J. ACM*, **19**, 675–698 (1972).
25. A. Nijholt, A survey of normal form covers for context free grammars', *Acta Inf.*, **14**, 271–294 (1980).
26. M. D. Mickunas, 'On the complete covering problem for LR(k) grammars', *J. ACM*, **23**, 17–30 (1976).
27. M. D. Mickunas and V. B. Schneider, 'A parser-generating system for constructing compressed compilers', *Comm. ACM*, **16**, 669–676 (1973).