CUCS-71-83

Covers of Attribute Grammars and Sub-Protocol Attribute Evaluators

by

Rodney Farrow

Computer Science Department
Columbia University
New York, New York 10027

12 September 1983

This research was partially supported by the Defense Advanced Research Projects Agency under contract number N00039-82-C-0427

Abstract

A terminology and notation, *semantically-trivial exact covers*, is introduced for describing a class of meaning-preserving transformations of attribute grammars. Two elements of this class, p-split and c-split, are studied in some detail and it is shown that for any non-circular attribute grammar G, p-split o c-split(G) is a *uniform* attribute grammar [25].

The class of uniform attribute grammars is of interest because particularly efficient attribute evaluators can be built for them: *straight-line evaluators*. Using these transformations to build a uniform grammar constructs a straight-line evaluator for the result as a side-effect. The class of uniform attribute grammars properly includes those attribute grammars that can be evaluated in left-to-right passes [2], alternating passes [10], sweeps, and also includes the class of ordered attribute grammars [14].

The straight-line evaluator for p-split(G) can be translated into a evaluator for G, which is not a straight-line evaluator but is nearly as efficient. This *protocol-evaluator* is compared to the evaluator of Kennedy and Warren [16] and Nielson's *direct evaluator* [20]. It can be viewed as an optimized, pre-compiled version of the latter, and is in some ways better and some ways worse than the Kennedy-Warren evaluator. From the comparison of the relative strengths and weaknesses of the protocol-evaluator and the Kennedy-Warren evaluator a new evaluator is derived, the *sub-protocol-evaluator*, which is more efficient than either.

Table of Contents

List of Figures

1. Introduction

Attribute grammars are an extension of context-free grammars that can be used to specify not only the valid context-free phrase structure of a language, but also an intended *translation* of strings in the language. For a particular string in the language, the process of computing the translation specified by an attribute grammar is refered to as *attribute evaluation* and an algorithm that implements this process is called an *attribute evaluator*. Many strategies for doing attribute evaluation, and their corresponding attribute evaluators, have been proposed [2, 10, 6, 16, 25, 14, 17]. Not all of these strategies will work on every attribute grammar; consequently, to each attribute evaluation strategy there corresponds a class of attribute grammars: namely, those attribute grammars that are amenable to evaluation by that strategy.

In this paper we will be concerned with an especially efficient class of evaluators that we call the *straight-line evaluators*. A straight-line evaluator is a *tree-walk evaluator* [16] in which each production of the grammar has a unique sequence of EVAL and VISIT instructions and this instruction sequence contains no conditional branches. The class of straight-line evaluators properly contains the left-to-right-pass evaluators of Bochmann [2], the alternating-pass evaluators of Jazayeri [10], the ordered evaluators of Kastens [14], and the sweep evaluators. It does not include the more general tree-walk evaluators of Kennedy-Warren [16] or Cohen-Harry [3]. A straight-line evaluator can be built for G iff G is a member of the class of *uniform* attribute grammars, defined by Warren [25].

We will show by construction how every non-circular attribute grammar can be transformed to a (larger) uniform attribute grammar. We call this construction the u-split transformation. It *splits* each non-terminal into several new non-terminals and then *lifts* each production to several new productions involving these split non-terminals. u-split is defined as the composition of two other transformations on attribute grammars: u-split = p-split o c-split. The c-split transformation takes an arbitrary attribute grammar and produces a corresponding *absolutely non-circular* attribute grammar [16]. The p-split transformation takes any absolutely non-circular attribute grammar and produces a corresponding uniform attribute grammar. The straight-line evaluator for the resulting uniform attribute grammar can be easily generated from the construction.

When u-split is the identity transformation on G it produces a straight-line evaluator for G. We compare this evaluator with the *ordered evaluators* of Kastens [14] and show that u-split can be viewed as a more effective extension of Kastens' strategy for constructing evaluators.

The last part of this paper uses the techniques of the p-split transformation to derive efficient attribute evaluators for those absolutely non-circular attribute grammars for which we are unable to build straight-line evaluators. The first such we describe is the protocol-evaluator. It can be viewed as an optimized, pre-compiled version of Nielson's *direct evaluator* and it is in some ways better and in some ways worse than the Kennedy-Warren evaluators. After comparing the relative strengths and weaknesses of these two strategies we derive the sub-protocol-evaluator: a combination of the protocol evaluator and the Kennedy-Warren evaluator that is more efficient than either.

2. Terminolgy and Definitions

A *context-free grammar* is a 4-tuple (N,Σ,S,P), where N is the set of *non-terminal* symbols, Σ is the set of *terminal* symbols, $S \in N$ is the *start symbol*, and P is the set of *productions*. A production is of the form $[p : X_0 ::= X_1 \cdots X_{np}]$. $X_0 \in N$ is the *left-part* of p; $X_1\ X_2 \ldots X_{np}$ is the *right-part* of p and for $i > 0$, either $X_i \in N$ or $X_i \in \Sigma$. Sometimes the expression "p[i]" is used to denote X_i.

Attribute grammars were first proposed by Knuth [18] as a way to specify the semantics of context-free languages. The basis of an attribute grammar is a context-free grammar. This describes the context-free language that is the domain of the translation, that is, those strings on which the translation is defined. This context-free grammar is augmented with *attributes* and *semantic functions*. Attributes are associated with the symbols of the grammar, both terminal and non-terminal. We write "X.A" to denote attribute A of symbol X, and $\mathcal{A}(X)$ to denote the set of attributes associated with X. Semantic functions are associated with productions; they describe how the values of some attributes of the production are defined in terms of the values of other attributes of the production.

Below is an attribute grammar that describes based integers in Ada and the values they denote. Examples of Ada based integers are: 16#9f = 9f base 16 = 159 base 10, or 2#10110 = 10110 base 2 = 22 base 10, or 3#10110 = 10110 base 3 = 93 base 10.

```
 1  number ::= digits1 '#' digits2.      /* P1 */
 2    number.VAL    = digits2.VAL
 3    digits2.RADIX = digits1.VAL
 4    digits1.RADIX = 10
 5    digits1.POWER = 1
 6    digits2.POWER = 1

 7  digits ::= digit.                    /* P2 */
 8    digits.VAL  = digit.VAL
 9    digit.POWER = digits.POWER

10  digits0 ::= digits1 digit.           /* P3 */
11    digits0.VAL   = digits1.VAL + digit.VAL
12    digits1.RADIX = digits0.RADIX
13    digits1.POWER = digits0.POWER * digits0.RADIX
14    digit.POWER   = digits0.POWER

15  digit ::= '0'.                       /* P4 */
16    digit.VAL = 0

17  digit ::= '1'.                       /* P5 */
18    digit.VAL = digit.POWER

19  digit ::= '2'.                       /* P6 */
20    digit.VAL = 2 * digit.POWER

          .
          .
          .

45  digit ::= 'F' | 'f'.                 /* P19 */
46    digit.VAL = 15 * digit.POWER
```

Figure 2-1: an attribute grammar for based numbers in Ada

Lines 1, 7, 10, 15, 17, ..., 45 of figure 2-1 are context-free productions; the other lines denote semantic functions. The notation of this example will be used throughout this paper: in the production of line 1, <digits1> and <digits2> denote separate occurrences of the same symbol, <digits>; the numeric suffixes distinguish these different occurrences. Different symbol-occurrences in a production, such as <digits> in production P1, give rise to different *attribute-occurrences.* Symbol-occurrences and attribute-occurrences are associated only with individual productions.

A semantic function specifies the value of an attribute-occurrence of the production, e.g. digits1.VAL. Semantic functions are pure functions, they have no side-effects. Their only arguments are either constants or other attribute-occurrences of the production.

How an attribute grammar specifies a translation can be most easily explained by an operational description. The underlying context-free grammar of an attribute grammar describes a language. Any string in this language has a parse tree associated with it by the grammar. The nodes of this parse tree can be labelled with symbols of the grammar. Each interior node of this tree has two productions associated with it. The left-part production (LP) is the production that derives this node. The children of this node are labelled with the symbols in the right-part of the production. The right-part production (RP) is the production that applies at the parent of this node, which is labelled with the left-part symbol of the RP production. This node and the siblings of this node are labelled with the symbols in the right-hand side of the RP production. Leaves of the tree don't have LP productions; the root doesn't have an RP production. Figure 2-2 shows a parse tree for the string 7#53. Each node in this tree is labelled with its associated grammar symbol, which is the left-part symbol of its LP production.

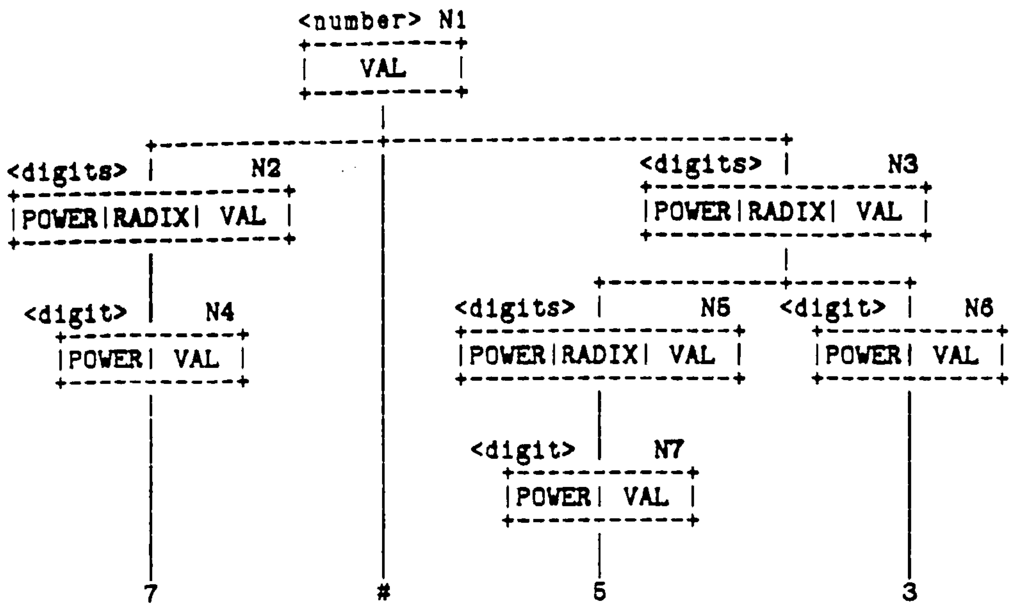


Figure 2-2: A semantic tree fragment

A *semantic tree* is a parse tree in which each node contains fields that correspond to the attributes of its labelling grammar symbol. Each of these fields is an *attribute-instance*. Associated with each attribute is a set of possible values that instances of this attribute can be assigned. This is analogous to the "type" of a variable in a programming language. However, each attribute-instance takes on precisely one such value; attribute-instances are not variables. The values of attribute-instances are specified by the semantic functions.

The semantic functions of a production represent a template for specifying the values of attribute-instances in the semantic tree. Consider figure 2-2 again. N3 is an semantic tree node that is associated with <digits2> in the production [number ::= digits1 '#' digits2] (its RP production) and N3 is associated with <digits0> in production [digits0 ::= digits1 digit] (its LP production). The semantic function `digits2.RADIX = digits1.VAL` indicates that the value of attribute-instance N3.RADIX will be copied from the value of attribute-instance N2.VAL. Similarly, the semantic function `digits0.VAL = digits1.VAL + digit.VAL` indicates that the value of attribute-instance N3.VAL should be calculated by adding together the values of N5.VAL and N6.VAL.

Since two different productions are associated with each attribute-instance, there could be two semantic functions that independently specify its value, one from the LP production and one from the RP production. If we assume that each attribute-instance is defined by only one semantic function, either from the LP production xor from the RP production, then we must guard against an attribute-instance not being defined at all because the LP production assumed that the RP production would define it and vice versa. These difficulties are avoided in attribute grammars by adopting the convention that for every attribute, X.A, either: (1) every instance of X.A is defined by a semantic function associated with its LP production, or (2) every instance of X.A is defined by a semantic function associated with its RP production. Attributes whose instances are all defined in their LP production are called *synthesized* attributes; attributes whose instances are all defined in their RP production are called *inherited* attributes. Every attribute is either inherited or synthesized. The start symbol has no inherited attributes; terminal symbols have no synthesized attributes. From the point of view of an individual production these conditions require that the semantic functions of a production MUST define EXACTLY the right-part occurrences of inherited attributes and all synthesized attributes of the left-part symbol. Inherited attributes propagate information down the tree, towards the leaves. Synthesized attributes propagate information up the tree, toward the root. The inherited attributes of a non-terminal X are denoted by $\mathcal{I}(X)$, the synthesized attriubtes by $\mathcal{S}(X)$: $\mathcal{A}(X) = \mathcal{I}(X) \uplus \mathcal{S}(X)$.

Thus the semantic functions of an attribute grammar specify a unique value for each attribute-instance. However, in order to actually compute the value of attribute-instance Z we must first have available the values of those other attribute-instances that are arguments of the semantic function that defines Z. In the example of figure 2-2, before N3.RADIX can be computed the value of N2.VAL must have already been computed. Such *dependency relations* restrict the order in which attribute-instances can be evaluated. In extreme cases an attribute-instance can depend on itself; such a situation is called a circularity and by definition such situations are forbidden from occuring in well-defined attribute grammars. In general, it is an exponentially hard problem [11] to determine that an attribute grammar is *non-circular*; i.e. that no semantic tree that can be generated by the attribute grammar contains a circularly defined attribute-instance. Fortunately there are several interesting and widely applicable sufficient conditions that can be checked in polynomial time [2, 10, 16, 14].

The result of the translation specified by an attribute grammar is realized as the values of one or more (necessarily synthesized) attribute-instances of the root of the semantic tree. In order to compute these values the other attribute-instances must be computed. An ATTRIBUTE EVALUATION PARADIGM is a meta-algorithm for building an algorithm that will compute attribute-instances in such an order that no attribute-instance is computed before all dependent attribute-instances are available and such that all attribute-instances of the root are computed. An attribute evaluation paradigm may work correctly only on a subset of all well-defined attribute grammars, but it must work correctly on any semantic tree of an acceptable attribute grammar.

Attribute grammars are attractive specification tools. Two principal reasons for this are their *locality of reference* and their *non-procedural nature*. We say that an attribute grammar has locality of reference in that the values it defines (i.e. the attribute-instances) are specified exclusively in terms of other attribute-instances local to a production. An attribute grammar does not contain any global variables or implicit state information that can affect the translation. Each local piece of an attribute grammar, i.e. each production, communicates with the rest of the attribute grammar only through strictly defined interfaces: the attributes of the symbols occurring in this production.

Like a context-free grammar, an attribute grammar is a description rather than an algorithm. Just as a context-free grammar specifies phrase-structure independently of a parsing algorithm, so does an attribute grammar specify semantics or translation without presuming an evaluation order. Because semantic functions are pure functions, the definition of an attribute-instance is determined by the attribute grammar and the semantic tree; not by the algorithms of the evaluator or those of the semantic functions. Thus, an attribute grammar is a locally described, non-procedural specification of values, rather than an algorithm for computing those values.

Although not universal (c.f. [6, 4, 12, 17]), the most widespread class of evaluators is the class of *tree-walk evaluators*. A tree-walk evaluator has a single locus of control that moves around an explicit semantic tree. The locus of control is always at some non-leaf of the tree. The evaluator executes a sequence of EVAL_f and VISIT_k instructions. An EVAL_f instruction says to invoke semantic function f of the production that applies at the current node, and use the resulting value to define the appropriate attribute-instance of either the current node or one of its children. Arguments supplied to this function are the (previously computed) values of appropriate attribute-instances of the current node and its children.

A VISIT_k instruction causes the evaluator's *locus of control* to move from the current node to either its parent or one of its children; VISIT_0 moves it to the parent, if $k > 0$ then VISIT_k moves the locus of control to the k-th child. The locus of control is never moved to a leaf; a VISIT_0 executed at the root indicates the end of attribute evaluation.

Tree-walk evaluators can also have further mechanisms to decide what sequence of instruction to execute at the different nodes of the semantic tree. For instance, the Kennedy-Warren evaluator [16] keeps a flag at each node that "remembers" what has been evaluated during earlier VISITs to the node, and, when

VISITing a child node, it passes a parameter that tells what new attributes have been computed since the last VISIT. Each time that a node N is VISITed from its parent the parameter of the VISIT and the flag that was left in the node are used to determine what sequence of instructions should be executed at N until the next VISIT_0 to N's parent. The new flag value to be left at N is also computed at the same time. A *PLAN* is Kennedy and Warren's name for the sequence of VISIT and EVAL instructions to be used during a VISIT to a node. On VISITing a node from its parent they determine which PLAN to use by means of a precomputed *GOTO-table*. In a Kennedy-Warren evaluator, each VISIT to a node from its parent results in the execution of a single PLAN at that node, and each PLAN is a straight-line sequence of VISIT and EVAL instructions. The only time a choice is made about what instructions to execute is when a PLAN is selected at the beginning of such a VISIT.

The Kennedy-Warren evaluator works only on the absolutely non-circular attribute grammars. In a later work [25] Warren described an extension of this evaluator, called the *coroutine evaluator* that works on <u>any</u> non-circular attribute grammar. This method allows a VISIT_0 instruction to pass an argument back to the parent of the current node, and for PLANs used at a node to select different instructions to execute based on the argument "returned" from a $\text{VISIT}_{k>0}$ instruction. Thus, unlike the Kennedy-Warren evaluator, the coroutine evaluator allows the sequence of instructions that are executed at a node to depend on the shape of the sub-trees that lie below this node. In a similar vein, Cohen and Harry [3] described an extension to the Kenedy-Warren evaluator that allows the evaluator to choose what sequence of instructions to execute based on flags left at the children of the current node. Like the coroutine evaluation strategy, Cohen and Harry's strategy allows the sequence of instructions executed at a node to depend on the structure of the sub-trees below that node. Hence Cohen and Harry's evaluator can also evaluate <u>any</u> non-circular attribute grammar.

In this paper we will be concerned with a class of tree-walk evaluator that is more restricted than the Kennedy-Warren evaluator. *Straight-line evaluators* are evaluators in which the sequence of instructions executed at a node depends only on which production applies at that node. Straight-line evaluators can leave a flag at each node that tells what has happened during previous VISITs to the node, but they do not pass arguments to VISIT instructions. For each production there is a single straight-line sequence of EVAL and VISIT instructions. The "flag" left at a node simply tells where in that sequence the evaluator was when it executed the most recent VISIT instruction. Evaluation then resumes at the next instruction in the sequence.

Several authors [25, 3, 14] have pointed out that tree-walk evaluators can be implemented by coroutines. Each production has an associated coroutine and there is a distinct instantiation of the appropriate coroutine at each interior node of the semantic tree. Here, VISIT instructions correspond to coroutine RESUMEs. In this model, the straight-line evaluators possess no explicit flags, and their coroutines consist of straight-line sequences of EVAL and VISIT (RESUME) instructions without any conditionally executed instructions.

Especially efficient straight-line evaluators can be built. Since no arguments are passed with VISIT instructions, no code need be compiled to implement this, or to use these values to select the appropriate instruction sequences to be executed. Because each production is translated into a single sequence of straight-line code, there is only one invocation of each semantic function. In contrast, the more general evaluators of Kennedy-Warren, Warren, and Cohen-Harry may have the same semantic function conditionally invoked several places in order to select the correct evaluation sequence to use at a node given its placement in some particular semantic tree. It is more feasible to compile in-line code for semantic function invocation when <u>several</u> copies of such code will <u>not</u> be needed; thus the evaluator will be faster.

The class of attribute grammars that can be evaluated by straight-line evaluators is fairly large. In [14] Kastens shows how evaluators for ordered attribute grammars can be implemented with one coroutine per production [14]. These coroutines contain no conditional instructions and and have exactly one invocation of each semantic function; hence they are straight-line evaluators. Evaluation in left-to-right passes [2], alternating passes [10], and sweeps is also straight-line evaluation.

3. Coverings of Attribute Grammars

Suppose $G = (N, \Sigma, S, P)$ and $\bar{G} = (\bar{N}, \Sigma, S, \bar{P})$ are two context-free grammars over Σ and that there exists an onto mapping (surjection), $\rho : \bar{N} \twoheadrightarrow N$ such that $[q : Y_0 ::= Y_1 \ldots Y_{nq} \in \bar{P}]$ implies $[p : \rho(Y_0) ::= \rho(Y_1) \ldots \rho(Y_{nq}) \in P.]$ Then the mapping $\rho : \bar{N} \twoheadrightarrow N$ can be extended to a mapping on productions, $\rho : \bar{P} \rightarrow P$. If ρ extended to productions has the property that:

for each production $p \in P$ and
  for each non-terminal $\bar{X} \in \rho^{-1}(p[0])$,
    $\rho^{-1}(p)$ contains exactly one production whose left-part is $\bar{X}$

then we say that $\bar{G}$ is an *exact cover* of G. [1] ρ is called the *covering map*.[2]

Theorem 1: If $\bar{G}$ is an exact cover of G then $L(\bar{G}) = L(G)$.

Since it is a surjection, the mapping ρ induces a partition both on the non-terminals of $\bar{G}$, $\{\rho^{-1}(X) \mid X \in N\}$, and on the productions of $\bar{G}$, $\{\rho^{-1}(p) \mid p \in P\}$. Each element of $\rho^{-1}(B)$ is called a *representative* of B, where B is either a non-terminal or a production. Furthermore, if $\rho(\bar{p}) = p$ we say that $\bar{p}$ is a *lifting of* p *into* $\bar{G}$.

Once the representatives for the non-terminals of G have been chosen an exact cover of G is determined by selecting which of the finitely many possible representatives of each production are to be included. At least one representative must be selected for each production in the covered grammar.

An exact cover of a context-free grammar is a very resticted modification and enlargement of that grammar. This can be seen by looking at respective parse trees for the same string. Suppose that $\bar{G}$ is an exact cover of G. Let T_G be the set of all derivation trees and sub-derivation-trees for G; that is, T_G is the set of all labelled trees, t, such that:

- the leaves of t are labelled with elements of Σ, and
- each interior node of t is labelled with a production $p \in P_G$, and
- for every interior node of t with label p, there are precisely np children of this node in t and the i-th child is labelled with p_i such that $p[1] = p_1[0]$.

Let $T_{\bar{G}}$ be the corresponding set of derivation trees and sub-trees for $\bar{G}$. The covering map ρ induces a mapping $\rho : T_{\bar{G}} \rightarrow T_G$. If $\rho(t_{\bar{G}}) = t_G$ then t_G and $t_{\bar{G}}$ have identically the same structure as trees; i.e., the same number of nodes, each node has the same number of descendents, etc. The only differences are the labels on the nodes that tell what production applies at this interior node.

Because $\bar{G}$ is an exact cover of G the mapping $\rho : T_{\bar{G}} \rightarrow T_G$ must be a surjection. If this map is also an injection (i.e., a one-to-one mapping) then we get the following.

[1] An exact cover of G is also a left cover of G and a right cover of G [1].

[2] We will let ρ denote not only the original covering map, but also its extension to productions and (later) its extension to derivation trees. The context of its use will always make clear which mapping we mean.

Theorem 2: If $\bar{G}$ is an exact cover of G, and the covering map extended to <u>full</u> derivation trees is an <u>injection</u> then $\bar{G}$ is ambiguous $\Rightarrow$ G is ambiguous.

Proof $\bar{G}$ is ambiguous implies that there exists two distinct derivation trees, $\bar{t}_1$ and $\bar{t}_2$, that derive the same sentence in the language. Since $\rho : T_{\bar{G}} \rightarrow T_G$ is injective it can not be that $\rho(\bar{t}_1) = \rho(\bar{t}_2)$ and hence these two elements of T_G are distinct derivation trees for the same sentence. Thus G is also ambiguous. Q.E.D.

The concept of exact covers has so far been defined only for context-free grammars. We want to extend this idea to attribute grammars, and we will do so in the simplest possible way.

If G and $\bar{G}$ are attribute grammars then $\bar{G}$ is a *semantically-trivial*, exact cover for G iff :

- the underlying context-free grammar for $\bar{G}$ is an exact cover of the underlying context-free grammar of G, with covering map ρ, and
- for each non-terminal $\bar{X}$ of $\bar{G}$, the attributes of $\bar{X}$ are identical with the attributes of $\rho(\bar{X})$, and
- for every production $\bar{p} \in \bar{G}$, ρ induces an isomorphism from the set of semantic functions of $\bar{p}$ to the set of semantic functions associated with $\rho(\bar{p}) \in G$; there is a one-to-one correspondence between these two sets of semantic functions, and if $\bar{X}_{1_0} = f(\bar{X}_{1_1}, \bar{X}_{1_2}, \ldots, \bar{X}_{1_2})$ is a semantic function of $\bar{p}$ then the corresponding semantic function of p is $\rho(\bar{X}_{1_0}) = f(\rho(\bar{X}_{1_1}), \rho(\bar{X}_{1_2}), \ldots, \rho(\bar{X}_{1_2}))$.

Theorem 3: If $\bar{G}$ is a semantically-trivial, exact cover for G, and $\bar{p} \in \bar{G}$ is a representative of production $p \in G$, then $D\bar{p}$, the dependency graph for $\bar{p}$, is isomorphic to Dp.

In this paper, we deal only with semantically-trivial, exact covers. Hereinafter, we will omit the *semantically-trivial* part and say only that *attribute grammar* $\bar{G}$ *is an exact cover of attribute grammar G*.

4. The c-split Transformation

The c-split transformation is a transformation that takes an attribute grammar G into an exact cover of G; $\bar{G}$ = `c-split(G)` implies that $\bar{G}$ exactly covers G. The representatives in $\bar{G}$ of a non-terminal X in G will correspond to the distinct *characteristic graphs* of X. The representatives in $\bar{G}$ of a production in G will be the obvious choice, as explained below.

A characteristic graph for a non-terminal X, is a directed graph, χ_X, whose nodes exactly correspond to the attributes of X and whose edges represent precisely the dependency relations among the attribute-instances of the root (an X-type node) of some semantic sub-tree. In general, each non-terminal may have several different characteristic graphs associated with it, reflecting the different dependencies that could be exhibited by different semantic trees. The sets of characteristic graphs of a non-terminal are computed by Knuth's corrected algorithm for determining circularity [18]. Characteristic graphs are also discussed by Cohen and Harry [3] and used in their attribute evaluation strategy.

c-split(G) is formed by splitting each non-terminal X into several non-terminals, one for each characteristic graph of X. The representatives, in $\bar{G}$, of a production $[p : X_0 ::= X_1 \cdots X_{n_p}]$ are all $\bar{G}$-productions of the form $[\bar{p} : \bar{X}_0 ::= \bar{X}_1 \ldots \bar{X}_{np}]$ such that $pr_0(Dp[\chi_{i_1}, \ldots, \chi_{i_{np}}]) = \chi_0$. Here $Dp[\cdots]$ denotes the dependency graph for production p augmented with the selected characteristic graphs, and pr_0 denotes projection onto the embedded graph for X_0.

This means that $\bar{G}$ will include precisely those productions, $\bar{p}$, such that the characteristic graph to which the $\bar{p}[0]$ corresponds is exactly the one induced by augmenting the dependency graph for $\rho(\bar{p})$ with the appropriate characteristic graphs of $\rho(\bar{p})[1]$, $\rho(\bar{p})[2]$, ...

c-split(G) can be exponentially larger than G since G can have exponentially many characteristic graphs [11]. However, if G is non-circular then c-split(G) is *absolutely non-circular* [16]; in fact, an even stronger statement can be made:

Theorem 4: Each non-terminal in c-split(G) has exactly one characteristic graph.

Proof is obvious by the construction.

The construction that comprises the c-split transformation has also been presented by Katayama [15], although not using the terminology presented here. There the attribute grammars whose non-terminals have a unique characterisic graph were called *simple*. The c-split transformation produces simple attribute grammars.

Simple grammars are stable under application of the c-split transformation: if G is simple then G = c-split(G). Unfortunately, the same is not true for absolutely non-circular grammars. Even if G is absolutely non-circular there may still be more than one characteristic graph for X, and hence more than one representative of X in c-split(G). However, the c-split transformation does preserve two important properties of the underlying context-free grammar: lack of ambiguity, and LR(k)-ness. It does not preserve LL(k)-ness, as is shown by the example of figure 4-1.

Lemma 1: If p is a production of G such that all right-part symbols of p are terminals, i.e., $p[1], p[2], \ldots p[np] \in \Sigma$, then there exists a unique production $\bar{p}$ in c-split(G) such that $\rho(\bar{p}) = p$.

Proof Since the right-part symbols of p are terminals, so must be the right-part symbols of any $\bar{p}$. These terminal symbols don't have any characterisic graph so the augmented dependency graph for $\bar{p}$ is determined only by the $D\bar{p}$, the dependency graph for the production. Since this is unique the left-part non-terminal, p[0], must also be unique; thus uniquely determining $\bar{p}$. Q.E.D.

Lemma 2: For every p in G, and $\bar{X}_1, \bar{X}_2, \ldots, \bar{X}_{np} \in N_{c\text{-}split(G)}$ there exists a unique $\bar{p} \in P_{c\text{-}split(G)}$ such that $\bar{p}[1] = \bar{X}_1$, $\bar{p}[2] = \bar{X}_2$, ..., $\bar{p}[np] = \bar{X}_{np}$, and $\rho(\bar{p}) = p$.

Proof According to the construction of c-split(G), only those productions $\bar{p}$ will be included in c-split(G) such that $\chi_0 = pr_0(Dp[\chi_1, \ldots, \chi_{np}])$, where χ_i is the unique characteristic graph corresponding to $\bar{p}[i]$ and pr_0 means projection onto the nodes corresponding to attribute-occurrences of X_0.

Thus, χ_0 is a function solely of the χ_i and $D\bar{p}$. χ_i is determined by choosing right-part $\bar{X}_i$, and $D\bar{p}$ is determined by p. Thus the left-part non-terminal, $\bar{p}[0]$ is completely determined and so is $\bar{p}$ itself. Q.E.D.

Theorem 5: The mapping $\rho : T_{c\text{-}split(G)} \rightarrow T_G$ is an injection.

Proof We must show that for every $t \in T_G$, there exists a unique $\bar{t} \in T_{c\text{-}split(G)}$ such that $\rho(\bar{t}) = t$. The proof is by induction on the height of tree t.

The basis step of this induction is when the height of t is 1. Let root(t be the root node of tree t, and let $p \in P_G$ be the label on root(t). Since the tree has height 1, the right-part non-terminals of p are all terminal symbols. By Lemma 4 there is only one production $\bar{p} \in P_{c\text{-}split(G)}$ such that $\rho(\bar{p}) = p$ and so $\bar{p}$ must be the label on the root of any tree $\bar{t}$ such that $\rho(\bar{t}) = t$. Since the labels of the leaves are fixed, this establishes the induction hypothesis when height is 1.

For the induction step, suppose that the hypothesis is true for any tree whose height is $\leq$ h and consider a tree t whose height is h+1. Let p be the label on the root of t; let $\bar{t}$ be such that $\rho(\bar{t}) = t$; let $\bar{p}$ be the root of $\bar{t}$; let $t_1, t_2, ..., t_{np}$ be the sub-trees that are the children of the root of t; and let $\bar{t}_1, \bar{t}_2, ..., \bar{t}_{np}$ be the sub-trees that are the children of the root of $\bar{t}$. By the induction hypothesis, the sub-trees $\bar{t}_1, \bar{t}_2, ..., \bar{t}_{np}$ are unique since we must have that $\rho(\bar{t}_i) = t_i$. It is now left to show that the production $\bar{p}$ is also uniquely defined.

But the production that labels the root of each $\bar{t}_i$ is well-defined and thus the right-part non-terminal symbols of $\bar{p}$ are determined. According to Lemma 4, since $\rho(\bar{p}) = p$, the production $\bar{p}$ is also uniquely determined. Since the label of the root of $\bar{t}$ is unique and the children of $\bar{t}$ are uniquely determined, it must be that $\bar{t}$ itself is uniquely determined. This established the inductive implication, and therefore the theorem. Q.E.D.

Theorem 6: If G is unambiguous then c-split(G) is unambiguous.

Proof follows immediately from Theorem 2 and the above Theorem 5.

Theorem 7: If G is LR(k) then c-split(G) is LR(k).

Proof follows by showing how a violation of the LR(k) condition in c-split(G) can be mapped into a violation of the LR(k) condition in G. Any sentential form of c-split(G) is mapped to a sentential form for G by the covering map ρ, as are right-most derivations of sentential forms. Following the LR(k) definitions and terminology of [1], suppose that there are two right-most c-split(G)-derivations:

$$S \underset{rm}{\overset{*}{\Rightarrow}} \bar{\alpha}\bar{A}w \underset{rm}{\overset{*}{\Rightarrow}} \bar{\alpha}\bar{\beta}w \qquad\qquad S \underset{rm}{\overset{*}{\Rightarrow}} \bar{\gamma}\bar{B}x \underset{rm}{\overset{*}{\Rightarrow}} \bar{\alpha}\bar{\beta}y$$

and that $FIRST_k(w) = FIRST_k(y)$. The two derivations are mapped by ρ to derivations in G:

$$S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\overset{*}{\Rightarrow}} \alpha\beta w \qquad\qquad S \underset{rm}{\overset{*}{\Rightarrow}} \gamma B x \underset{rm}{\overset{*}{\Rightarrow}} \alpha\beta y$$

and still $FIRST_k(y) = FIRST_k(x)$. Since G is LR(k) it must be that $\alpha = \gamma$, $A = B$, and $x = y$. This means that $\rho(\bar{\alpha}) = \rho(\bar{\gamma})$, and $\rho(\bar{A}) = \rho(\bar{B})$. But the covering map ρ preserves the length of a sentential form, and so it must be that the length of $\bar\alpha$ is the same as the length of $\bar\gamma$ and hence that $\bar\gamma = \bar\alpha$.

Thus $\bar\gamma\bar{B}x = \bar\alpha\bar{B}y \underset{rm}{\Rightarrow} \bar\alpha\bar\beta y$ and the c-split(G)-production $[p2 : \bar{B} ::= \bar\beta]$ is used in the final right-most derivation step. Consequently, the two c-split(G) productions $[p1 : \bar{A} ::= \bar\beta]$ and $[p2 : \bar{B} ::= \bar\beta]$ are such that $\rho(p1) = \rho(p2)$ and $p1[1] = p2[1], \ldots, p1[l] = p2[l]$. Thus, by the previous lemma 4, $p1 = p2$ and $\bar{A} = \bar{B}$. Q.E.D.

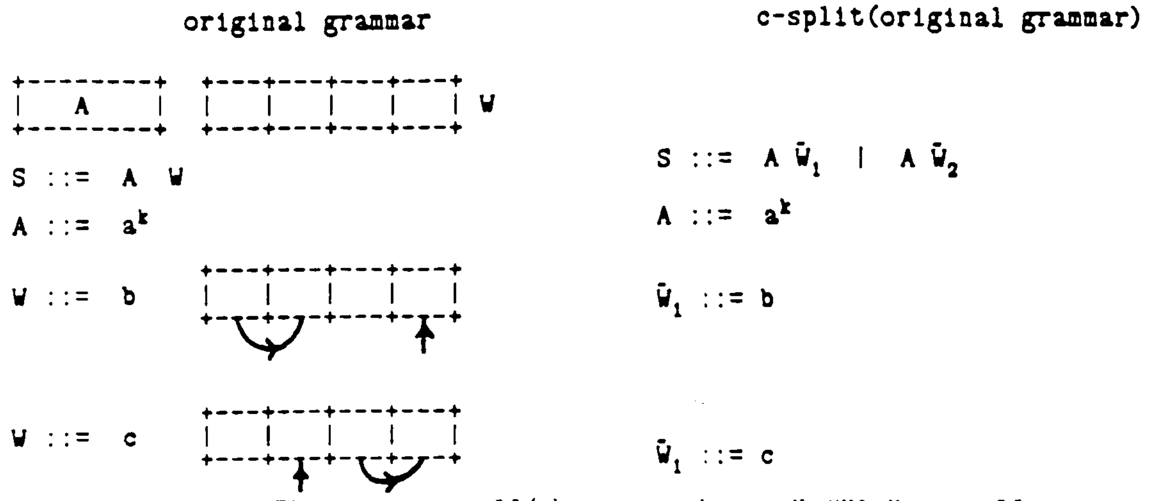


Figure 4-1: An LL(1) grammar that c-split "lifts" to non-LL

As we shall see in the next section, the reason we use c-split is to ensure that we have an absolutely non-circular grammar; we don't need a *simple* grammar.

5. The p-split transformation

The p-split transformation is defined only on absolutely non-circular attribute grammars. It is similar to the c-split transformation in that it lifts an attribute grammar G to an exact cover of G. Where the c-split transformation splits a non-terminal into representatives that correspond to characteristic graphs, the p-split transformation splits non-terminals into representatives that correspond to *protocols*.

Protocols[3] were introduced by Warren [25] as a tool for studying his *coroutine* evaluators. A protocol $\pi = \pi(1), \ldots, \pi(lp)$ for a non-terminal X is a sequence of non-empty sets of attributes such that:

- there are an even number of elements in the sequence,
- the elements of the sequence constitute a partition of the attributes of X,
- even elements of the sequence consist entirely of synthesized attributes, and odd elements of the sequence consist entirely of inherited attributes.

A protocol for a non-terminal X can be interpreted as a plan for how a tree-walk evaluator should evaluate the attributes of X. Each pair of protocol elements $\pi(2j-1)$, $\pi(2j)$ represents a "visit" to a subtree derived from X. The inherited attributes of $\pi(2j-1)$ are the attributes made newly available as "inputs" to the next visit; the synthesized attributes in $\pi(2j)$ represent the "outputs" produced as a result of the visit.

If $[p : X_0 ::= X_1 \cdots X_{np}]$ is a production then a set of protocols $\{\pi_i \mid 0 \leq i \leq np\}$ is *consistent for* p iff a tree-walk attribute evaluator fragment for p can be pieced together so that the π_i summarize the

[3] a protocol is called a "(reduced) partition of attributes" in [8]

order in which attributes are computed and made available across the "borders" of the non-terminals of p. This idea is made more precise shortly. By definition [25] an attribute grammar is <u>uniform</u>[4] iff

> for each non-terminal X there exists a unique protocol π_X, such that
>
> for each production $[p : X_0 ::= X_1 \cdots X_{np}]$,
>
> $\{\pi_{X_i} \mid 0 \leq i \leq np\}$ is consistent for p.

In this case we say that $\{\pi_X \mid X \in V_G\}$ is a *consistent <u>set</u> of protocols for G*. G is uniform iff G has a consistent set of protocols.

Warren [25] gives a polynomial-time algorithm for determining if a given set of protocols is consistent. However, Engelfriet and Filé show [5] that it is an NP-complete problem to determine whether an attribute grammar has a consistent set of protocols, i.e. whether an attribute grammar is uniform. Warren's algorithm for deciding whether a set of protocols was consistent for p was given as a special case of a more general and complex algorithm. We will describe a simpler algorithm, based on Engelfriet and Filé's, and use it to introduce some useful terminology.

Suppose that b is an inherited attribute of non-terminal X and that c is a synthesized attribute of X, and that $b \in \pi_X(i)$, $c \in \pi_X(i+k)$. This means that b is always going to be evaluated <u>before</u> c, and hence b <u>can</u> <u>not</u> depend on c. A set of protocols will be consistent for production p if such constraints embodied in the protocols are consistent with the usual dependencies of p. In order to determine this we use the concept of a *protocol graph*.

A protocol graph is like an IO-graph or a characteristic graph in that it represents a non-circular relation among the attributes of a non-terminal. In a protocol graph an edge $b \rightarrow c$ represents that attribute b occurs before attribute c in the protocol, and that consequently, b will be evaluated before c. Compare this with a characteristic graph, where an edge $b \rightarrow c$ means that in some semantic sub-tree, c *depends on* b. The difference is that b may be evaluated before c even though there is no dependency relation between b and c.

Definition 1: $\delta = (N_\delta, E_\delta)$ is a protocol graph for non-terminal X iff:

- N_δ, the nodes of δ, correspond to attributes of X,
- δ is non-circular,
- for every pair of attributes $I \in \mathcal{I}(X)$, $S \in \mathcal{S}(X)$, either $I \overset{*}{\rightarrow} S$ in δ, or $S \overset{*}{\rightarrow} I$ in δ.

Note that since δ is an acyclic graph it can not be that <u>both</u> $I \overset{*}{\rightarrow} S$ <u>and</u> $S \overset{*}{\rightarrow} I$. Also, it is quite allright for a protocol graph to have paths $I_1 \overset{*}{\rightarrow} I_2$ even if there is no S such that $I_1 \overset{*}{\rightarrow} S$ and $S \overset{*}{\rightarrow} I_2$; similarly, it is allright to have paths $S_1 \overset{*}{\rightarrow} S_2$ even if there is no I such that $S_1 \overset{*}{\rightarrow} I$ and $I \overset{*}{\rightarrow} S_2$.

Definition 2: If δ is a protocol graph for X and π is a protocol for X, then δ is *consistent with* π iff $b \in \pi(i)$, $c \in \pi(i+k)$ implies $b \overset{*}{\rightarrow} c$ in δ.

Theorem 8: If δ_X is a protocol graph for non-terminal X then there exists a unique protocol π such that δ_X is consistent with π.

> **Proof** : δ determines that $\pi(1)$ must be all inherited attributes b such that there is no path from a synthesized attribute to b; and that $\pi(lp)$ must be all synthesized attributes c such that there is no path from c to an inherited attribute. Then $\pi(1)$ and $\pi(lp)$ and edges incident upon them can be deleted from δ and this process repeated to find $\pi(2)$ and $\pi(lp-1)$, ... δ will not be consistent with any other protocol. Q.E.D.

[4]uniform attribute grammars are called "simple multi-visit" in [5]

For each protocol π_X there is a *canonical protocol graph*, δ_X, whose edges are: $\{(A,B) \mid A \in \pi_X(i) \text{ and } B \in \pi_X(i+1) \text{ and } 0 \leq i \ \ lp\}$. This graph is a subgraph of every protocol graph for X.

As with IO graphs, we can use protocol graphs to augment dependency graphs for productions: $Dp[\delta_{p[0]}, \delta_{p[1]}, ..., \delta_{p[np]}]$ The set of protocols is consistent for production p iff $Dp[\delta_{p[0]}, \delta_{p[1]}, ..., \delta_{p[np]}]$ is acyclic [5].

The p-split transformation applied to G builds an exact cover of G in which the representatives of a non-terminal X correspond to protocols for X. p-split will lift a production $[p : X_0 ::= X_1 \cdots X_{np}]$ in G to a production $[\bar{p} : \bar{X}_0 ::= \bar{X}_1 \ldots \bar{X}_{np}]$ such that the protocols $\{\pi_i\}$ to which the $\{\bar{X}_i\}$ correspond are consistent for p.

Definition 3: A *consistent* *collection* *of protocols* for an attribute grammar G is a set of sets of protocols, $\{ \Pi_X \mid \Pi_X$ is a set of protocols for $X \in N_G\}$ such that

```
for every production $[p : X_0 ::= X_1 \cdots X_{np}] \in P_G$
    for every protocol $\pi_0 \in \Pi_{X_0}$ there exist protocols
        $\pi_1 \in \Pi_{X_1}, \pi_2 \in \Pi_{X_2}, \ldots, \pi_{np} \in \Pi_{X_{np}}$
      such that $\{\pi_0, \pi_1, \pi_2, \ldots, \pi_{np}\}$ is consistent for p
```

The non-terminals of p-split(G) will correspond to the members of the Π_X. The start symbol S lifts to exactly one representative $\bar{S}$, which corresponds to the protocol $(\emptyset, \{A(S)\})$. A production $[p : X_0 ::= X_1 \cdots X_{np}]$ lifts to all productions $[\bar{p} : \bar{X}_0 ::= \bar{X}_1 \ldots \bar{X}_{np}]$ such that:

1. each $\bar{X}_i$ represents a protocol $\pi_i \in \Pi_{\rho(\bar{X}_i)}$, and
2. $\{\pi_i \mid 0 \leq i \leq np\}$ is consistent for p, and
3. if $\rho(\bar{p}_1) = \rho(\bar{p}_2)$ then $\bar{p}_1[0] \neq \bar{p}_2[0]$

That every attribute grammar can be *lifted* to a uniform attribute grammar was also noticed by Engelfriet and Filé [5] (of course, they don't use this notation). Their construction builds an exact cover that is always exponentially larger than the original grammar. In this construction, "representatives of non-terminal X" include all possible protocols for X, and, generally, each production *lifts* to many more representatives than it does in the p-split construction. One consequence is that their *lifted* attribute grammar will usually be ambiguous even though the original was not. p-split(G) preserves both lack of ambiguity and LL(k)-ness. Unfortunately, p-split does not preserve LR(k)-ness, as is shown by figure 5-1.

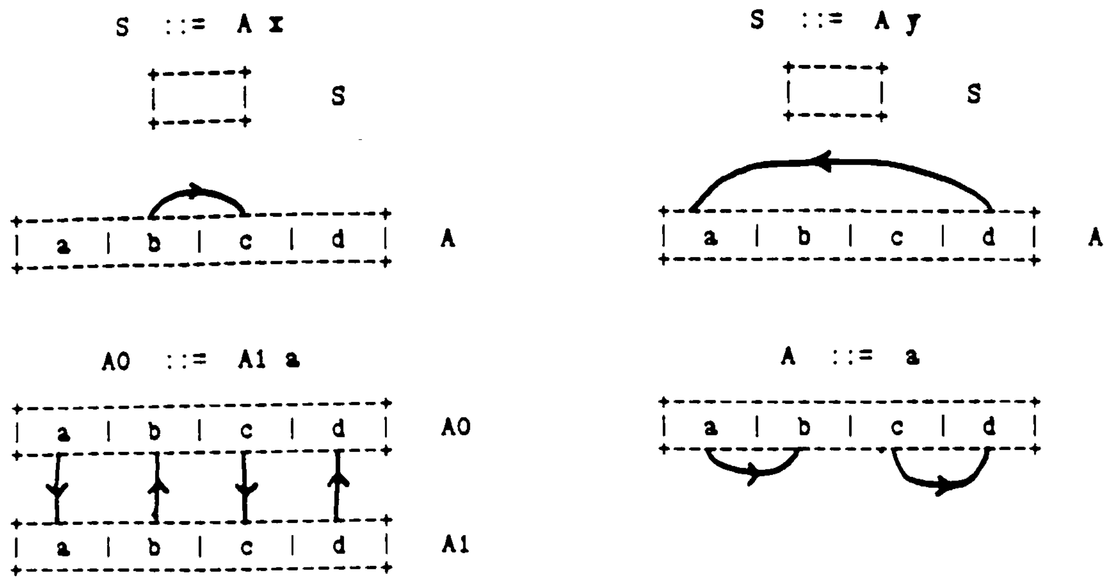
Lemma 3: For each production $[p : X_0 ::= X_1 \cdots X_{np}]$ there exists a unique production $[\bar{p} : \bar{X}_0 ::= \bar{X}_1 \ldots \bar{X}_{np}]$ such that $\rho(\bar{X}_0) = X_0$ and $\rho(\bar{p}) = p$.

Proof : this result is explicitly guaranteed by point (3) of the construction of p-split productions.

Theorem 9: If G is LL(k) then p-split(G) is LL(k).

Proof follows by showing how a violation of the LL(k) condition in p-split(G) can be mapped into a violation of the LL(k) condition in G. Any sentential form of p-split(G) is mapped to a sentential for for G by the covering map ρ, as are left-most derivations of sentential forms. Following the LL(k) definitions and terminology of [1], suppose that there are two left-most p-split(G)-derivations:

$$S \overset{*}{\underset{lm}{\Rightarrow}} w\bar{A}\bar{\alpha} \underset{lm}{\Rightarrow} w\bar{\beta}\bar{\alpha} \overset{*}{\underset{lm}{\Rightarrow}} wx \qquad\qquad S \overset{*}{\underset{lm}{\Rightarrow}} w\bar{A}\bar{\alpha} \underset{lm}{\Rightarrow} w\bar{\gamma}\bar{\alpha} \overset{*}{\underset{lm}{\Rightarrow}} wy$$



Nodes that are instances of non-terminal A must be evaluated either according to protocol ({A.a}, {A.b}, {A.c}, {A.d}) or according to protocol ({A.c}, {A.d}, {A.a}, {A.b}). The p-split transformation will "lift" A to either X or Y, corresponding to the first or second of these protocols, respectively. The resulting grammar, p-split(G), is given below. It is not LR(k) for any k because there is a conflict between reducing the first a to either an X or a Y.

S ::= X x	S ::= Y y
X ::= X a | a	Y ::= Y a | a

Figure 5-1: An LR(0) grammar, G, for which p-split(G) can not be LR(k) for any k.

such that $\text{FIRST}_k(y) = \text{FIRST}_k(x)$. Suppose further that $\bar{\beta} \neq \bar{\gamma}$. The two p-split(G)-derivations are mapped by ρ to G-derivations:

$$S \overset{*}{\underset{lm}{\Rightarrow}} wA\alpha \overset{*}{\underset{lm}{\Rightarrow}} w\beta\alpha \overset{*}{\underset{lm}{\Rightarrow}} wx \qquad\qquad S \overset{*}{\underset{lm}{\Rightarrow}} wA\alpha \overset{*}{\underset{lm}{\Rightarrow}} w\gamma\alpha \overset{*}{\underset{lm}{\Rightarrow}} wy$$

such that $\text{FIRST}_k(y) = \text{FIRST}_k(x)$. Since G is LL(k) it must be that $\beta = \gamma$, and hence $\rho(\bar{\beta}) = \rho(\bar{\gamma})$. Thus the two p-split(G) productions $[p1 : \bar{A} ::= \bar{\beta}]$ and $[p2 : \bar{A} ::= \bar{\gamma}]$ are such that $\rho(p1) = \rho(p2)$ and $p1[0] = p2[0]$. Thus, by the previous lemma 8, $p1 = p2$ and $\bar{\beta} = \bar{\gamma}$. Q.E.D.

Lemma 4: For every $t \in T_G$ and protocol $\pi \in \Pi_{\text{root}(t)}$, there exists a unique $\bar{t} \in T_{\text{p-split}(G)}$ such that: $\rho(\bar{t}) = t$ and, root($\bar{t}$) corresponds to π.

Proof: Let $p \in P_G$ be the label on root(t) and $\bar{p}$ be the label on root($\bar{t}$). It must be that $\rho(\bar{p}) = p$. According to lemma 8, $\bar{p}$ is completely determined.

The proof is by induction on the height of tree t. The basis step of this induction is when the height of t is 1. Since the tree has height 1, it consists entirely of nodes incident on production $\bar{p}$ and hence is uniquely determined by it.

Suppose now that the lemma is true for all trees whose height is $\leq$ h and that t has height h+1. Let $\{t_i\}$ be the sub-trees of t whose roots are children of root(t), and let $\{\bar{t}_i\}$ be the sub-trees of $\bar{t}$ whose roots are children of root($\bar{t}$). Each t_i is a subtree whose height is $\leq$ h, and each $\bar{t}_i$ is a subtree whose height is $\leq$ h. Since $\rho(\bar{t}) = t$ it must be that $\rho(\bar{t}_i) = t_i$.

The production $\bar{p}$ uniquely determines which p-split(G) non-terminals are associated with the the roots of the subtrees $\{\bar{t}_i\}$ and so, by the induction hypothesis all of these subtrees are uniquely determined. Since $\bar{t}$ consists only of the subtrees $\{\bar{t}_i\}$ and the production that labels root($\bar{t}$), and since each of these is unqiue it follows that $\bar{t}$ itself is unique. Q.E.D.

Theorem 10: The mapping $\rho : T_{p\text{-split}(G)} \rightarrow T_G$ is an injection, when it is restricted to full derivation trees of G.

Proof follows from the above lemma 9. p-split(G) contains a unique representative of S, the start symbol of G, and for any full derivation tree of G the root must be labelled with a production whose left-part is S.

Theorem 11: If G is unambiguous then p-split(G) is unambiguous.

Proof follows immediately from Theorem 2 and the above Theorem 10.

Our construction of a sufficient collection of protocols is based on an iterative closure algorithm similar to Knuth's algorithm for building characteristic graphs. With each non-terminal X we will associate a set of protocols Π_X. Initially these sets will be empty, except that $\Pi_S = \{(\emptyset, A(S))\}$, where S is the start symbol of the grammar. Protocols are added according to the following rule:

```
repeat until no more protocols are added
    for each production  [p : $X_0$ ::= $X_1 \cdots X_{np}$]
        for each protocol $\pi_0 \in \Pi_{X_0}$
            based on $\pi_0$ and the dependencies of p,
              compute protocols $\pi_{X_1}, \pi_{X_2}, \ldots, \pi_{X_{np}}$,
              such that $\{\pi_{X_0}, \pi_{X_1}, \pi_{X_2}, \ldots, \pi_{X_{np}}\}$ is consistent for p
            for i $\in$ [1..np]
                if $\pi_i \notin \Pi_{X_i}$ then add $\pi_i$ to $\Pi_{X_i}$
            endFor
        endFor
    endFor
endRepeat
```

This algorithm is called the *protocol closure algorithm*. It terminates because there are finitely many distinct protocols for a non-terminal and no protocol is ever added to a Π_{X_i} more than once. The key step is to compute the protocols for $X_1, X_2, \ldots, X_{np}$ from the protocol for X_0. That this can be done for any absolutely non-circular attribute grammar was shown by Nielson [19], who also describes a straightforward procedure for doing so. Her procedure is one possible implementation of the general *protocol propagation* algorithm described below. The protocol propagation algorithm implements a function $P_G \rightarrow (\Pi_{p[0]} \rightarrow (\Pi_{p[1]} \times \ldots \times \Pi_{p[np]})$. If protocol propagation maps production p and left-part protocol π_0 to right-part protocols $\pi_1, \ldots, \pi_{np}$ then the corresponding production will be a representative of p in p-split(G). If protocol propagation was a relation instead of a function then Lemma 8 would not hold.

The protocol propagation algorithm begins by taking the dependency graph for p, Dp, and augmenting it with δ_0, the protocol graph for the left-part non-terminal, and with the IO graphs for the right-part non-terminals. These IO graphs are available because we require the attribute grammar to be absolutely non-circular. The resulting $Dp[\delta_0, IO_{p[1]}, ..., IO_{p[np]}]$ may induce some further dependencies among the attributes of right-part non-terminals because the protocol graph δ_0 may contain more constraints than does $IO_{p[0]}$. However, this graph will be acyclic because δ_0 is a <u>protocol</u> <u>graph</u> and contains IO_{X_0} within it.

From $Dp[\delta_0, IO_{p[1]}, ..., IO_{p[np]}]$ we can build protocols for each of the right-part X_i by non-deterministically adding edges between previously unrelated (inherited,synthesized) pairs of attributes of the same right-part non-terminal. We keep adding edges until the subgraph corresponding to each right-part non-terminal becomes a protocol graph (Definition 7). Then, according to Theorem 8, each of the right-part subgraphs determines a unique protocol for their corresponding right-part non-terminal. Adding an edge between previously unrelated nodes may introduce new constraint on the dependencies of this production. These effects must be determined before we choose another edge to add so that whenever we add a new edge it truly is between heretofore <u>unrelated</u> nodes.

A variety of specific algorithms can be realized by using some more specific policy to narrow the freedom to add edges. One such policy is to finish building a protocol for one right-part non-terminal before starting to construct the protocol for any other right-part non-terminals. Further policies can be devised to tell us how to add edges between a particular non-terminal's attributes in order to build a protocol graph. This latter process we refer to as *protocolizing* the graph.

One policy to use in protocolizing a graph is that an attribute should be evaluated as soon as possible; that is, each attribute should be put in the earliest possible element of the protocol. This may be thought of as *greedy* protocolizing. Figure 5-2 shows an algorithm that implements this policy. Another policy is for the evaluation of each attribute to be delayed until just before it is needed; i.e. each attribute should go in the latest possible element of the protocol. We can call this *lazy* protocolizing. The algorithm of figure 5-3 does *lazy* protocolizing. To see the difference between these two policies look at figure 5-4. This shows dependencies among attributes of a non-terminal on the left and three possible protocols that would be valid for these dependencies. One of these protocols is the result of *greedy* protocolizing; one is the result of *lazy* protocolizing; the third is a protocol different from either of those.

With these definitions we can easily characterize Nielson's algorithm, mentioned on page 14, for computing right-part protocols. It is a combination of two policies: protocolize one right-part non-terminal at a time (in left-to-right order), and use the "greedy" algorithm to protocolize each non-terminal.

In the worst case, the protocol closure algorithm can build a collection of protocols whose size is exponential in the number of attributes of a non-terminal. Specifically, the number of protocols in Π_X can be exponential in the number of attributes of X. We would like to design the protocol closure algorithm so that it always finds a minimal collection of protocols. Unfortunately, this is an intractable problem. If the grammar is uniform then each of the Π_X will be a singleton set, yet finding such sets is an NP-complete problem (mentioned earlier).

A backtracking algorithm could be designed to find a minimal collection of protocols by exhautive search, but this could be exorbitantly expensive. What we expect is that an acceptably small collection can be found in a reasonable amount of time by using appropriate heuristics. In the rest of this section we will present our heurstics and modify the protocol propagation algorithm to reflect them. This revised algorithm will always construct a consistent collection of protocols; we expect that it will construct a

```
Let $N_X$ = the nodes corresponding to attribute-occurrences of X in Dp[ ... ]
For every A $\in N_X$ define NUM(A) as:
  NUM(A) =

    if P(A) = $\emptyset$ then    0
              else 1 + max{ NUM(B) | B $\in P_A$ }
    fi
    where
      P(A) = if A is inherited then
               {B  |  B $\in N_X$  and  B $\xrightarrow{+}$ A $\in$ Dp[ ... ]  and  B is synthesized}
             else /* A is synthesized */
               {B  |  B $\in N_X$  and  B $\xrightarrow{+}$ A $\in$ Dp[ ... ]  and  B is inherited}
             fi
    /*
      NUM(A) is the maximum number of subpaths on any dependency path ending
      at node A; which subpaths have sources that represent inherited attributes of A
      and targets that represent synthesized attributes of X, or vice versa.
    */

repeat
  Let F = {(D,C) | NUM(D) > NUM(C), and
                   D $\xrightarrow{+}$ C $\notin$ Dp[ ... ], and
                   either C is inherited   and D is synthesiszed, or
                          C is synthesized and D is inherited
          }

   if  F $\neq \emptyset$  then

     Choose (D,C) $\in$ F such that  (B,A) $\in$ F  implies NUM(C) $\leq$ NUM(A)
       /*
         This condition says that we will choose to add an edge into a node that is
         minimal with respect to its NUM value.
       */

      Add (D,C) to the graph embedded within Dp[ ... ]
       /*
         Note that adding this edge does not disturb the value of NUM(C) because
         the length of any new path to C thus created is at most 1 + NUM(D),
         which by hypothesis is no greater than NUM(C).  Thus the NUM values
         need not be recalculated.

         However, adding this edge may well add a path between other, previously
         unrelated, nodes of $N_X$ so F must be recomputed.  Of course, the cost of
         this can be kept manageable by using some sort of finite differencing
         optimization.
        */

    fi

  until F = $\emptyset$
```

Figure 5-2: "Greedy" protocolizing algorithm

small collection.

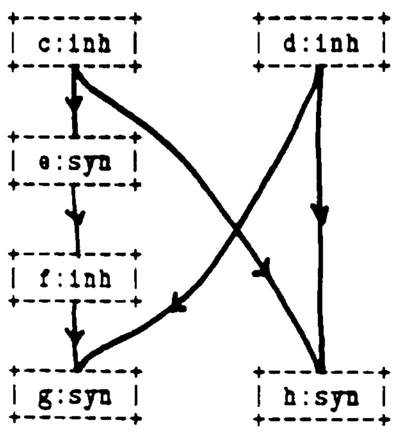
We will use the following heurstics, listed in order of precedence:

1. never add a protocol to the collection unless the collection is currently not consistent,
2. when adding a protocolizing dependency (X.A, X.B), if there are two or more occurences of X in the production then add edges between the attributes of all occurences of X so long as this does not introduce a circularity; if it does introduce a circularity then don't add this edge unless every other candidate edge is in the same situation,
3. if a new protocol must be constructed try first to replace an existing protocol with a *refinement* of itself.

In general these heuristics will not completely determine the right-part protocols based on the dependencies of the production and the left-part protocol. Additional heuristics or just blind guessing will be needed to form a deterministic algorithm. Additional heuristics might be designed to satisfy goals other than minimizing the number of protocols; e.g. to minimize the run-time storage needed by the

```
Let $N_X$ = the nodes corresponding to attribute-occurrences of X in Dp[ ... ]
For every $A \in N_X$ define NUM(A) as:
  NUM(A) =

    if P(A) = Ø then    0

              else 1 + max{ NUM(B) | B ∈ $P_A$ }
    fi
    where
      P(A) = if A is inherited then
                {B  |  B ∈ $N_X$  and  A $\xrightarrow{+}$ B ∈ Dp[ ... ]  and  B is synthesized}
             else /* A is synthesized */
                {B  |  B ∈ $N_X$  and  A $\xrightarrow{+}$ B ∈ Dp[ ... ]  and  B is inherited}
             fi
    /*
      NUM(A) is the maximum number of subpaths on any dependency path starting
      at node A; which subpaths have sources that represent inherited attributes of A
      and targets that represent synthesized attributes of X, or vice versa.
    */

repeat
  Let F = {(D,C) | NUM(D) > NUM(C), and
                   D $\xrightarrow{+}$ C ∉ Dp[ ... ], and
                   either C is inherited   and D is synthesiszed, or
                          C is synthesized and D is inherited
          }

   if  F ≠ Ø  then

     Choose (D,C) ∈ F such that  (B,A) ∈ F  implies NUM(D) ≤ NUM(B)
       /*
          This condition says that we will choose to add an edge out of a node that is
          minimal with respect to its NUM value.
       */

      Add (D,C) to the graph embedded within Dp[ ... ]
       /*
          Note that adding this edge does not disturb the value of NUM(D) because
          the length of any new path from D thus created is at most 1 + NUM(C),
          which by hypothesis is no greater than NUM(D).  Thus the NUM values
          need not be recalculated.

          However, adding this edge may well add a path between other, previously
          unrelated, nodes of $N_X$ so F must be recomputed.  Of course, the cost of
          this can be kept manageable by using some sort of finite differencing
          optimization.
        */

    fi

  until F = Ø
```

Figure 5-3: "Lazy" protocolizing algorithm



```
Result of ''greedy'' protocolizing
          ({c,d}, {e,h}, {f}, {g})

Result of ''lazy'' protocolizing
          ({c}, {e}, {d,f}, {g,h})

      A third distinct protocol
          ({c,d}, {e}, {f}, {g,h})
```

Figure 5-4: Attribute dependencies and some possible protocols

attribute evaluator.

Our first heuristic is quite straight-forward. Suppose that a new protocol π was added to Π_X and we are calculating what will be the corresponding protocols for non-terminals in the right-part of production p if π is the protocol of p[0]. The heuristic says that all possible combinations of existing protocols of right-part symbols should be checked before we construct yet another protocol. If most right-part non-terminals have only one protocol this is reasonably inexpensive. However, if there are k right-part non-terminals and each of them has only 2 protocols there are 2^k combinations to check. We advocate the use of this potentially expensive heuristic because we expect that in common attribute grammars most non-terminals will need only one protocol.

The second heuristic serves as a check on whether a certain protocolizing edge should be added. It is designed to discover some of those edges that, if added, must cause two different protocols to be used for X.

The third heuristic introduces protocol *refinements*.

Definition 4: A *direct refinement* of a protocol $\pi = (\pi(1), ..., \pi(k), \pi(k+1), ..., \pi(n))$, is a new protocol $\pi' = (\pi'(1), ..., \pi'(k), \pi'(k+1), \pi'(k+2), \pi'(k+3), ..., \pi'(n+2))$ such that for some odd k, $0 \leq k$ n,

1. $\pi(i) = \pi'(i)$ for $i<k$, and
2. $\pi(i) = \pi'(i+2)$ for $i>k+1$, and
3. $\pi(k) = \pi'(k) \cup \pi'(k+2)$, and
4. $\pi(k+1) = \pi'(k+1) \cup \pi'(k+3)$.

Intuitively, a direct refinement of a protocol corresponds to splitting one VISIT to a sub-tree into two consecutive VISITS, and leaving the rest of the protocol unchanged. $\pi(k)$ is the set of inherited attributes that is split, $\pi(k+1)$ is the set of synthesized attributes. A *refinement* of a protocol is the result of forming a sequence of direct refinements.

Definition 5: π' is a *refinement* of π iff: (1) π' is a direct refinement of π, or (2) there exists a protocol π'' such that π'' is a direct refinement of π and π' is a refinement of π''.
This recursive definition is well-founded because direct refinement always increases the length of a protocol by 2, and the maximum length of any protocol for X is the number of attributes for X.

The reason that refinements are interesting is that we need not use both a protocol and some refinement of it in evaluating an attribute grammar; only the refinement is necessary. This is established by the following lemmas and theorem. First, though, let us introduce some notation that will make these arguments let verbose.

If $\pi(X)$ is a protocol for non-terminal X and $b \in \mathcal{A}(X)$ such that $b \in \pi(i)$ then we say that i is the *index of b in π* and denote this by $i_\pi(b)$.

Lemma 5: Suppose that π, π' are protocols for X and that π' is a direct refinement of π formed by splitting $\pi(i),\pi(i+1)$ into $\pi'(i),\pi'(i+1),\pi'(i+2),\pi'(i+3)$. For every pair of attributes $b,c \in \mathcal{A}(X)$, if $i_\pi(b) < i_\pi(c)$ then either:
1. $i_{\pi'}(b) < i_{\pi'}(c)$, or
2. $b \in \pi(i)$, $c \in \pi(i+1)$, $b \in \pi'(i+2)$, and $c \in \pi'(i+1)$, and hence $b \in \mathcal{I}(X)$, $c \in \mathcal{S}(X)$.

Proof is obvious by inspection.

Lemma 6: Let p be a production in Bochmann Normal Form. Suppose that $\pi_{p[0]}, \pi_{p[1]}, ..., \pi_{p[np]}$ is a consistent set of protocols for p. For every path $b \rightarrow c$ in $Dp[\delta_0, \delta_{p[1]}, ..., \delta_{p[np]}]$ such that $b,c \in \delta_{p[i]}$ it must be that $i_\pi(b) < i_\pi(c)$.

Proof It must be the case that either $i_\pi(c) < i_\pi(b)$, or $i_\pi(c) = i_\pi(b)$, or $i_\pi(c) > i_\pi(b)$.

The proof establishes that the first two conditions are not possible.

If $i_\pi(c) < i_\pi(b)$ then there would be a path $c \overset{*}{\rightarrow} b$ in $\delta_{p[i]}$, and this together with the path of the hypothesis would constitute a cycle. Thus the first of the above conditions is not possible.

If $i_\pi(c) = i_\pi(b)$ then both b and c must belong to the same protocol element hence they are either both inherited or both synthesized. In this case, we may assume without loss of generality that the path $b \overset{*}{\rightarrow} c$ consists of only one edge; any such path must contain only nodes lying in the same protocol element and hence this path consists entirely of such edges. If these nodes are synthesized then this will not happen because $\delta_{p[i]}$ is the canonical protocol graph for $\pi_{p[i]}$; if they are inherited it will not happen because the production is in Bochmann Normal Form and hence there are no Dp-edges from one right-part, inherited attribute-occurrence to another.

Thus it must be that $i_\pi(b) < i_\pi(c)$. Q.E.D.

Theorem 12: Suppose that G is an attribute grammar in Bochman Normal Form, and that $\{\Pi_Y \mid Y \in V_G\}$ is a consistent collection of protocols for G, and that Π_X contains two protocols, π_X and π'_X such that π' is a refinement of π. Let Π'_X be the set of protocols for X that results when π is deleted from Π_X. Then $\{\Pi_Y \mid X \neq Y\} \cup \Pi'_X$ is also a consistent collection of protocols for G.

Proof . We will establish the theorem for the case when π' is a direct refinement of π; the general case then follows by induction.

We must show that for every production p, and for every protocol $\pi_0 \in \Pi_{p[0]}$ there exist protocols $\pi_i \in \Pi_{p[i]}$ such that $\{\pi_0, ..., \pi_{np}\}$ is consistent for p. Since π, the replaced protocol, is not an element of the new Π'_X we needn't worry about those productions where $p[0] = X$. However, where X is a right-part non-terminal of p we must show that if π is replaced by π' then the resulting set of protocols is still consistent for p. Furthermore, we will assume that X occurs only once in the right-part of p; if X appears more than once the following argument is not invalidated - simply use induction on the number of occurrences of X in the right-part of p.

Suppose that $X = p[i]$ so that $\{\pi_0, ..., \pi_i{=}\pi, ..., \pi_{np}\}$ is consistent for p; we must show that $\{\pi_0, ..., \pi_i{=}\pi', ..., \pi_{np}\}$ is also consistent for p. Let $D = Dp[\delta_0, ..., \delta_i{=}\delta, ..., \delta_{np}]$, $\bar{D} = Dp[\delta_0, ..., \delta_i{=}\bar{\delta}, ..., \delta_{np}]$. The hypotheses of the theorem imply that D is acyclic; we need to show that $\bar{D}$ is also acyclic.

Let $\bar{D} - \bar{\delta}$ be the graph obtained by deleting from the augmented dependency graph, any edges in the protocol graph. $\bar{D} - \bar{\delta}$ is a subgraph of D, and so any path that lies entirely within $\bar{D} - \bar{\delta}$ must be acyclic. Therefore, any cycle in $\bar{D}$ must contain an edge from $\bar{\delta}$.

Thus, it will suffice to show that for any pair of nodes $b,c \in \bar{\delta}$, if there is a path $b \overset{*}{\rightarrow} c$ in $\bar{D}$ then $i_{\pi'}(b) < i_{\pi'}(c)$. This would preclude the possiblity of there being a cycle in $\bar{D}$ that contained an edge from $\bar{\delta}$. If the above condition holds for two paths $b \overset{*}{\rightarrow} c$ and $c \overset{*}{\rightarrow} d$ then it must also hold for the concatenation of the two paths, $b \overset{*}{\rightarrow} c \overset{*}{\rightarrow} d$. Consequently, we can restrict our attention to only those minimal paths $b \overset{*}{\rightarrow} c$ in $\bar{D}$ such that $b,c \in \bar{\delta}$. Such minimal paths are either edges of $\bar{\delta}$, or they are paths in $\bar{D} - \bar{\delta}$. Clearly, edges of $\bar{\delta}$ must conform to the above condition.

Since $\bar{D} - \bar{\delta}$ is a subgraph of D, any path $b \overset{*}{\rightarrow} c$ in $\bar{D} - \bar{\delta}$ is also a path in D, and so by lemma 11, $i_{\pi}(b) < i_{\pi}(c)$. We need to show that this inequality also holds for the respective indices in the refined protocol π'. Lemma 11 says that either: $i_{\pi'}(b) < i_{\pi'}(c)$, or that b represents an inherited attribute and c represents a synthesized attribute. However, c can not represent a synthesized attribute because the path $b \overset{*}{\rightarrow} c$ contains no edges of the protocol graph and hence can not contain any edge whose target is c.

Therefore it must be that $i_{\pi'}(b) < i_{\pi'}(c)$. Q.E.D.

This theorem says that, if we must add a new protocol to some Π_X, and if it suffices to add π', a refinement of some protocol π already in Π_X, then we can replace π with π' and so not have to increase the size of our sufficient collection of protocols. Notice, however, that we must still propagate the new protocol π' through all productions that have X as their left-part. This propagation of π' can result in new protocols being added, which new protocols can in turn give rise to still other protocols, etc.

6. The u-split Transformation

The u-split transformation is the composition of p-split with c-split, `u-split = p-split o c-split`. The c-split transformation takes any non-circular attribute grammar to a simple attribute grammar. Since a simple attribute grammar is an absolutely non-circular attribute grammar the p-split transformation is defined on the result of c-split and takes it to a uniform attribute grammar. Furthermore, since neither c-split nor p-split introduce ambiguity we have:

Theorem 13: If G is unambiguous then u-split(G) is unambiguous.

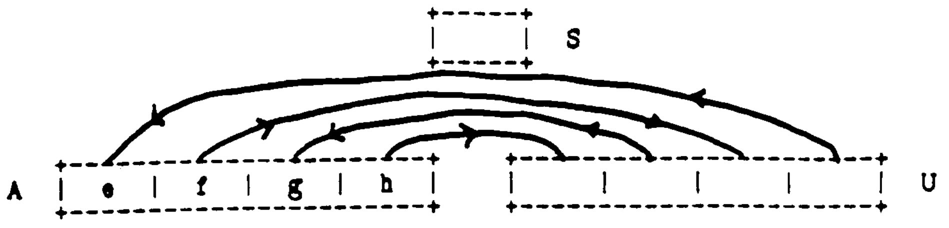
However, c-split need not preserve LL(k)-ness, and p-split need not preserve LR(k)-ness, so an easily parsed grammar can be "lifted" to one that is hard to parse. Figure 6-1 gives an example of this.

The complexity of c-split and p-split is exponential in the number of attributes per non-terminal. Fortunately the c-split transformation does not change the number of attributes per non-terminal so there is no 2-level exponentiality effect. That is, if each non-terminal of G has at most k attributes and there are at most n non-terminals then c-split(G) will contain at most $n*2^k$ non-terminals but each of these will still have no more than k attributes. Therefore, u-split(G) can contain at most $(n*2^k)*2^k$ non-terminals, each of which has at most k attributes. Nonetheless, the "lifted" attribute grammars can contain exponentially many non-terminals and productions. Hopefully, this worst-case performance will not occur for common attribute grammars.

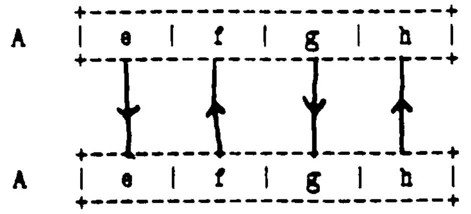
General tree-walk evaluators (e.g. [16, 3]) can select different evaluation orders depending on the structure of the given semantic tree they are evaluating. In contrast, a straight-line evaluator must always evaluate attributes in a certain, prescribed order. The u-split transformation transfers to the parsing algorithm the responsibility for determining in advance how to evaluate the attributes of a particular semantic tree.

Recall that the reason we were interested in uniform attribute grammars was that particularly simple evaluators could be built for them, straight-line evaluators. The u-split transformation gives us a way to build a straight-line evaluator for any attribute grammar: construct u-split(G) and then build the straight-line evaluator for u-split(G), using the protocols for non-terminals that were found during the construction of u-split(G). Having the protocols around is clearly important since finding a consistent set of protocols is NP-complete.

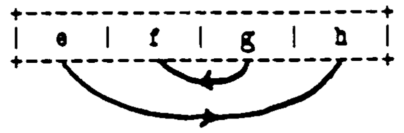
S ::= A U



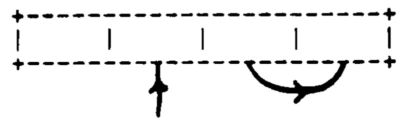
A ::= A a



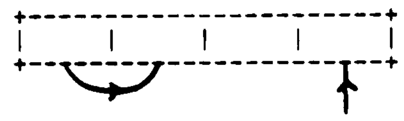
A ::= a



U ::= c



U ::= d



c-split(G) is:

S ::= A C

S ::= A D

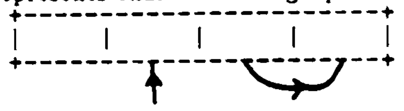
A ::= A a | a

C ::= c

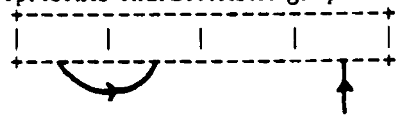
D ::= d

where ρ(C) = ρ(D) = U, and

C represents characteristic graph



D represents characteristic graph



u-split(G) = p-split(c-split(G)) is:

S ::= X C

S ::= Y D

X ::= X a | a

Y ::= Y a | a

C ::= c

D ::= d

where ρ(X) = ρ(Y)= A, and

X represents protocol ({e},{f},{g},{h})

Y represents protocol ({g},{h},{e},{f})

Figure 6-1: G is LL and LR but u-split(G) is neither

At least two problems could arise in doing this:

- u-split(G) might be so big that a straight-line evaluator for it is not feasible,
- we may not be able to parse u-split(G) deterministically, from left to right.

In the next section we discuss some measures we can take to avoid these problems and we compare our resulting attribute evaluator(s) with others that have appeared in the literature.

7. Building attribute evaluators

As figure 6-1 shows, the interaction of c-split and p-split can transform a grammar that was easy to parse into one it is impractical to parse. However, if the original grammar G was absolutely non-circular then we need not build c-split(G) at all; just build p-split(G). The class of absolutely non-circular attribute grammars is large and most AG-based translator-writing systems [22, 9, 13, 8] process a strict (sometimes very strict) subset of this class. Not forming c-split(G) if it's not needed also keeps manageable the size of the grammar on which p-split operates.

When G is not absolutely non-circular then we must form c-split(G). Such a process is expensive, increases the size of the resulting evaluators, and may not preserve the LL-ness of G. However, the LR-ness of G is preserved, and this process is clearly no worse than simply refusing to build an evaluator for G. Further research into why attribute grammars fail to be absolutely non-circular may lead to good heuristics that can be incorporated into the c-split construction so as to lift grammars to smaller absolutely non-circular grammars.

Regardless of whether we must first use c-split to get an absolutely non-circular attribute grammar, the core of our strategy is building protocols. If the protocol propagation algorithm builds a consistent set of protocols then: (1) the attribute grammar is uniform, (2) p-split(G) $\Rightarrow$ G, and (3) we can construct a straight-line evaluator for G from these protocols. There is good reason to believe that this can be done for many useful attribute grammars. Those attribute grammars for which the protocol propagation algorithm finds a consistent set of protocols is closely related to the class of *ordered* attribute grammars [14].

In defining the ordered attribute grammars and showing how to construct evaluators for them [14], Kastens was really describing an algorithm to compute a protocol for each non-terminal; this is the calculation of the EDS relation. The class of ordered attribute grammars was then defined to be those such that, for each production p, the particular set of protocols found by the ordering algorithm was consistent for p. Protocol propagation improves on this method by using more information and being more flexible in building protocols.

p-split is defined only on absolutely non-circular attribute grammars because the IO-graphs are needed in order to propagate protocols. However, Kastens understood that if an attribute grammar is uniform then an even stronger condition than absolute non-circularity is necessary. This condition, similar to absolute non-circularity, is:

- for each non-terminal X, IDS_X, a graph on (or relation among) the attributes of X is computed via an iterative closure algorithm; $IO_X \subseteq IDS_X$,
- then it is required that, for each production p, $Dp[IDS_{X_0}, IDS_{X_1}, ..., IDS_{X_{np}}]$ must be acyclic.

Let us call a grammar that satisfies this condition a regular attribute grammar. p-split's protocol propagation algorithm can be modified to use IDS graphs rather than IO graphs; let p-split$'$ be this modified version of p-split. The class of attribute grammars on which p-split$'$ is the identity is a superset of the class of ordered attribute grammars; if G is ordered then `p-split'(G) = G`.

There are attribute grammars that are regular, but not ordered, and which p-split$'$ *lifts* to themselves; see figure 7-1, an example taken from [14]. The computation of the EDS relations by Kastens' ordering algorithm corresponds to *protocolizing* each non-terminal independently with the "lazy" protocolizing algorithm (see figure 5-3, section). p-split$'$ is more effective than the ordering algorithm at finding a consistent set of protocols because:

1. the protocol propagation algorithm adds protocolizing dependencies incrementally and propagates their effects throughout a production before adding any more, and
2. heuristics, such as protocol refinement, are used to try to minimize the number of protocols per non-terminal.

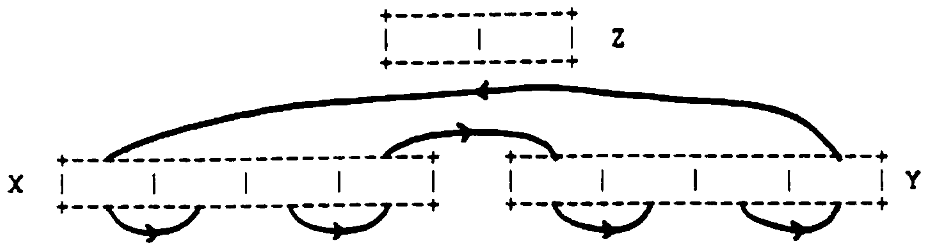


Figure 7-1: G is regular but not ordered and G = p-split(G)

Kastens claims [14] that the class of ordered attribute grammars is sufficiently large to contain most of the grammars people naturally write to describe programming languages. Still, there are regular grammars that are not uniform (figure 7-2), and absolutely non-circular grammars that are not regular. It seems unnecessarily severe to refuse to build an evaluator because one non-terminal requires two protocols. An obvious solution in such a case is to construct the straight-line evaluator for p-split(G); if there aren't many protocols in the consistent collection of protocols then p-split(G) won't be very large and neither will its straight-line evaluator. The problem with this is that it can destroy the LR-ness of the original grammar. Fortunately, we can arrange matters so that the source input is parsed according to the original grammar G, but the attribute evaluation is done by the straight-line evaluator for p-split(G).

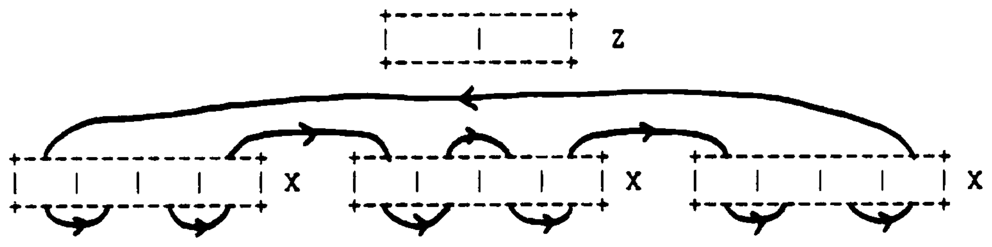


Figure 7-2: G is regular but not uniform

We will translate a sentence in L(G) as follows:

1. parse the sentence according to grammar G and building a derivation tree, $t \in T_G$,
2. translate t to $\bar{t} \in T_{p\text{-split}(G)}$, where $\rho(\bar{t}) = t$,
3. use the straight-line evaluator for p-split to compute the translation of t.

Steps (1) and (3) are straight-forward and need no elaboration. Step (2) can be done because the mapping $\rho : T_{p\text{-split}(G)} \rightarrow T_{p\text{-split}(G)}$ is an injection on full derivation trees (Theorem 10), section . The procedure for translating t to $\bar{t}$ is a top-down application of lemma 9 of section . Furthermore, it turns out that step (2) can be automatically subsumed in step (3) and no explicit translation need ever take place.

In section 2 we described how how straight-line evaluators could be implemented as coroutines. However, they can also be realized as subroutines: Kastens [14] describes how to build ordered evaluators as subroutines; building subroutines to implement alternating-pass evaluators is described in [7, 8]. For each production, the sequence of instructions that comprise the straight-line evaluator at that production can be partitioned into consecutive sub-sequences by the VISIT_0 instructions so that each sub-sequence is terminated by a VISIT_0 instruction, and so that each sub-sequence contains no embedded VISIT_0 instruction. Each such sub-sequence is the list of instructions that will be executed at a node during one of the VISITs to that node from its parent.

Since G is uniform, there is only one protocol that can be used for each non-terminal X, and this protocol determines: (1) how many VISITs will be made to each instance of X, and (2) what attributes will be computed during those VISITs. Thus, if p_1 and p_2 are two distinct productions such that $p_1[0] = X = p_2[0]$ then their respective evaluation sequences must each be partitioned into the same number of concatenated subsequences. Each of these subsequences corresponds to a consecutive pair of elements of the protocol for X, $\pi(2i\text{-}1)$, $\pi(2i)$. Similarly, if production p has X as its k-th right-part non-terminal then the sequence of instructions for p will contain a subsequence of VISIT_k instructions, one instruction for each pair of elements in the protocol for X.

Straight-line evaluators are implemented as a set of subroutines: one subroutine for each VISIT to a non-terminal. More specifically, for each non-terminal X in the grammar, create a separate subroutine for each pair $\pi_X(2i\text{-}1)$, $\pi_X(2i)$. Call these subroutines *VISIT-procedures.* In each production's sequence of instructions, replace every VISIT_k with an invocation of the appropriate VISIT-procedure.

Each VISIT-procedure for X consists of a case statement that has one case-clause for each production that could apply at X. A VISIT-procedure is invoked at a semantic tree node and, based on what production applies at that node, one of the case-clauses is executed. The case-clause that corresponds to a production p (whose left-part is X) is one of those sub-sequences of instructions for p which are terminated by a VISIT_0 instruction; what part of π_X this VISIT-procedure represents determines which of those sub-sequences to use.

Note that the VISIT-procedure version of a straight-line evaluator does not require that a flag be left at each node to summarize the current state of attribute evaluation at that node. This doesn't mean that this information is no longer needed. Rather, it is encoded in the dynamic state of the evaluator - the stacked return addresses of unfinished VISIT-procedure-instances for ancestors of the current node.

If we implement the straight-line evaluator for p-split(G) via VISIT-procedures then we need not explicitly translate the semantic tree for G into a semantic tree for p-split(G). Each VISIT-procedure is designed for a particular non-terminal of its grammar. A VISIT-procedure needs to know what production applies at some node only in order to choose which case-clause to execute. But for every pair $X \in N_G$, $\tilde{X} \in \rho^{-1}(X)$, there is a one-to-one correspondence between those p-split(G)-productions, $\tilde{p}$, such that $\tilde{p}[0] = \tilde{X}$ and those G-productions, p, such that $p[0] = X$. Therefore, the VISIT-procedures for p-split(G) can be constructed to use the productions of G rather than the more numerous productions of p-split(G). A VISIT-procedure never needs to know what non-terminal labels a semantic tree node; this information is implicitly represented by the state of the attribute evaluator; that is, which VISIT-procedure is currently executing.

Thus the VISIT-procedure version of the straight-line evaluator for p-split(G) can be built to work on a semantic tree built according to the grammar G rather than p-split(G). The input can be parsed by the grammar for G, rather than by the more complicated grammar for p-split(G); but the straight-line evaluator for p-split(G) can still be used for attribute evaluation. We call this "hybrid" evaluator the *protocol-evaluator* for G.

8. Comparison of the protocol-evaluator with other evaluators

8.1. Direct evaluators

Protocol-evaluators are identical in many respects with the *direct evaluators* of Nielson [20]. Nielson realized that protocols could be assigned to semantic tree nodes in a top-down sweep over the semantic tree and that this process could be overlapped with the top-down operation of the attribute evaluator. The protocol-evaluator improves on the direct evaluator by being smaller. If the protocol-evaluator for G was built from a consistent collection of protocols that included every protocol for every non-terminal of G then the result would be essentially the same as the direct evaluator for G. The protocol closure algorithm and protocol propagation algorithm together precompute a limited set of protocols that is sufficient for evaluating any semantic tree of the attribute grammar. In contrast, Nielson's direct evaluator is prepared to use any possible protocol of each non-terminal. Precomputing protocols not only allows us to use heuristics to look for efficient evaluators (i.e., small collections of protocols), it also makes it feasible to generate evaluators with such optimizations as in-line code for semantic functions, attribute stacks [23, 21], and static subsumption [8].

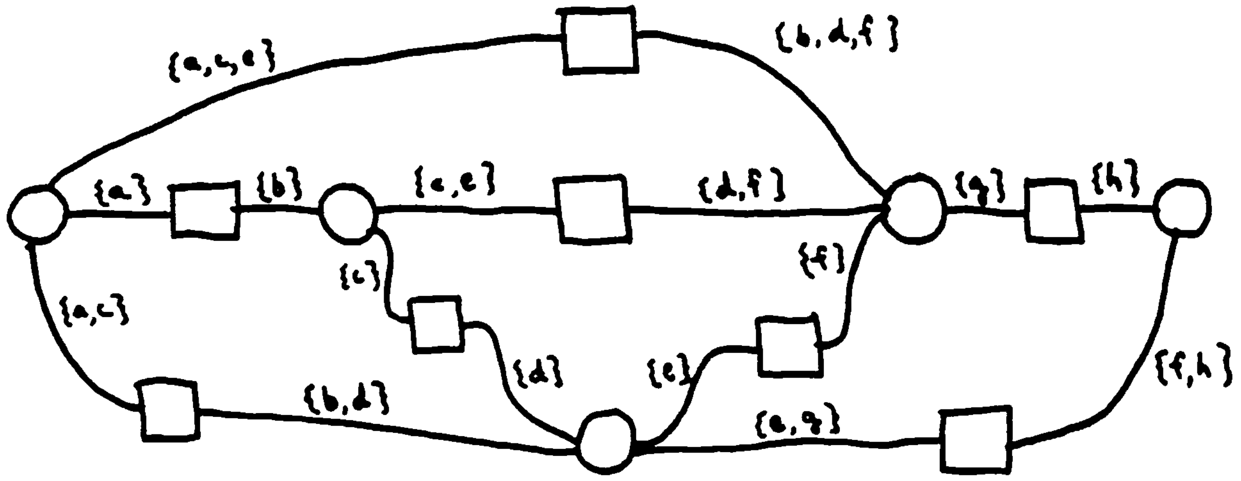
8.2. Kennedy-Warren evaluators

Protocol-evaluators are different in many respects from the tree-walk evaluators of Kennedy and Warren [16] (herinafter refered to as KW-evaluators). A KW-evaluator leaves some "state" information at each interior node of the semantic tree. This state information is used on successive VISITS to the node to determine what attributes should then be evaluated. When a KW-evaluator VISITs a sub-tree it knows what synthesized attributes of the root of that sub-tree will be available after the VISIT. However, it doesn't know which of those attributes will be evaluated on this VISIT and which have already been evaluated on previous VISITs. This is because the current node may have been previously VISITed itself by any of several different sequences of VISITs, each of which would have made different sets of attributes available, and each of which could have induced a different VISIT-sequence on this node's children.

On the other hand, a protocol-evaluator keeps no state information in the nodes, except what production applies at a node. A protocol-evaluator knows precisely what sequence of VISITs has been made to children of the current node because it knows precisely what sequence of VISITs has been made to the current node. The entire sequence of VISITs that has been, and will be, made to the current node is part of the "state" of the protocol-evaluator. Consequently, the protocol-evaluator knows not only what will be available <u>after</u> VISITing a sub-tree, but also exactly which attributes will be computed <u>during</u> the VISIT. This information can be useful. For example, to implement either the "attribute stacks" [23] or "static subsumption" [8] storage optimization, the evaluator must know precisely when an attribute is evaluated. The extra information stored by the KW-evaluator isn't superfluous, though; it is encoded in the state (and stack) of the protocol-evaluator itself. Thus, space in every node of the semantic tree is traded off against the size of the evaluator and its dyanamic storage requirements (i.e., stack space).

This doesn't mean that the KW-evaluator is "inferior" to the protocol evaluator; it just means that a protocol-evaluator may be much larger than a KW-evaluator. Consider the *history graph* used to build the KW-evaluator. There is one connected component of the history graph for each production, and each component has an initial node and a final node. Each node of the history graph represents a "state" that the evaluator could be in when VISITing a semantic tree node at which this production applies. An edge from node A to node B in the history graph means that the KW-evaluator could have been in first state A and then state B on two successive VISITs to a semantic tree node at which this production applies. Remember that a protocol is the sequence of sets of attributes that are evaluated just prior to and during successive VISITs. Thus, a protocol corresponds to a *maximal path through the history graph*. If all paths through the history graph are really needed to evaluate this attribute grammar then the number of states in the protocol-evaluator will be the same as the number of maximal paths through the history graph. In the worst-case, the number of maximal paths could be exponential in the number of nodes of the history graph.

Figure 8-1 shows a history graph for a production. Below it are all the protocols that correspond to maximal paths through the history graph.



({a},{b},{c,e},{d,f},{g,h})
({a},{b},{c},{d},{e},{f},{g,h})
({a},{b},{c},{d},{e,g},{f,h})
({a,c},{b,d},{e},{f},{g,h})
({a,c},{b,d},{e,g},{f,h})
({a,c,e},{b,d,f},{g},{h})

Figure 8-1: A history graph and the protocols corresponding to its maximal paths

Another difference between the KW-evaluator and the protocol-evaluator is the way in which they are built. The protocol-evaluator is built top-down, in a goal-directed manner. New protocols are added to the consistent collection of protocols only when they are needed to achieve some specific [sub-]goal, i.e. to implement the VISIT-sequence of a production for some left-part protocol. A new protocol is added only when it is a necessary component of a specific solution of a [sub-]goal, and this specific solution is completely known when the new protocol is added. Finally, the protocol itself becomes a sub-goal.

In contrast, the history graph construction of Kennedy and Warren's algorithm is a bottom-up process. A history graph is incrementally extended by adding new states and edges whenever it is determined that a particular state could be entered with a new parameter value. There is no other attempt to take into account either the previous sequence of VISITs to this node or, more importantly, the later VISITs that will be made to this node. The construction terminates only when no more states or edges need to be added to the history graph. Commenting on an extension of the history graph construction [25], Warren says

> " The main fault of the constructions so far considered is the absence of any overall planning during construction of an evaluator. The [history graphs] grow up little by little in a decentralized way, and as a result there may be an undesirable amount of diversity in the end product. ... ".

As an illustration of the effect of this difference between top-down and bottom-up construction, the last protocol listed in figure 8-1 is sufficient by itself to evaluate this production. The protocol closure and protocol propagation algorithms of p-split find this consistent collection of protocols because they are using heuristics, such as protocol refinement, to "look for" a sufficient, but small, collection of protocols.

9. Sub-protocol-evaluators

In this section we use our insight into the relative strengths of the protocol-evaluator and KW-evaluator to define an evaluator that is more efficient than either. This new *sub-protocol-evaluator* is the result of a sort of *common sub-expression elimination* applied to protocol-evaluators.

Remember that each protocol for a non-terminal X gives rise to a number of VISIT-procedures. If there are two protocols π_1, $\pi_2 \in \Pi_X$ then there will be be two distinct sets of VISIT-procedures. However, some of the VISIT-procedures for π_1 will be identical with VISIT-procedures for π_2. This will happen whenever precisely the same attributes of X are available before the two VISITs, and precisely the same attributes of X are computed during the two VISITS. We can detect this condition just by examining the two protocols. If $\pi_1(1) \cup \pi_1(2) \cup \ldots \cup \pi_1(2j-1) = \pi_2(1) \cup \pi_2(2) \cup \ldots \cup \pi_2(2k-1)$ and $\pi_1(2j) = \pi_2(2k)$ then the VISIT induced by $(\pi_1(2j-1),\ \pi_1(2j))$ will be identical to the VISIT induced by $(\pi_2(2k-1),\ \pi_2(2k))$. Each VISIT-procedure can be characterized by the set of attributes that have been computed prior to the VISIT, and the set of synthesized attributes that will be computed during the VISIT.

Since these two VISITs are identical we don't need both of their associated VISIT-procedures; a single copy suffices. The sub-protocol-evaluator is derived from the protocol-evaluator by eliminating any such multiple copies of VISIT-procedures. Invocations of duplicated VISIT-procedures are replaced with invocations of a single, *canonical* VISIT-procedure. This VISIT-procedure is determined by the set of attributes already available and the set of synthesized attributes made newly available by the VISIT. Having several different protocols give rise to a single VISIT-procedure is similar to having more than one entry arc into a single entry node of the history graph for a KW-evaluator. Recall that the ability of a state of the history graph to belong to more than one maximal path is the reason that the protocol-evaluator could potentially be much larger than the KW-evaluator.

The sub-protocol-evaluator can be viewed as resulting from two modifications to the KW-evaluator. One of these modifications is the removal of the "state" flag left at a node and the GOTO-table. When constructing the plan for a VISIT we know what state each right-part non-terminal will be in when a VISIT is scheduled to it; this state is needed in order to compute the yield of the VISIT and in order to make the corresponding entry in the history graph. Thus the process of taking this "state" and the input parameter value and "looking up" a plan in the GOTO-table can be done at evaluator-construction-time rather than at evaluation-time.

The penalty paid for this optimization is that an attribute-instance is never evaluated earlier than our most pessimistic calculation of when it can be evaluated. The evaluator is unable to detect and take advantage of fortuitous situations where some attribute-instances could be evaluated earlier than was calculated. This is not much of a penalty because the planning algorithm (c.f. the evaluator itself) depends on the pre-calculated yield of a scheduled VISIT and hence is unable to take advantage of such fortuitous occurrences; additional VISITs will be scheduled even though the desired attributes have already been evaluated.[5]

The second modification to the KW-evaluator concerns the use of collections of protocols as goals to use in building PLANs. The KW planning algorithm scheduled VISITs at random to any sub-tree with a non-empty yield. In contrast, protocol propagation is very goal-directed; striving to find a small but consistent collection of protocols. Furthermore, protocol propagation introduces a new VISIT only in the context of a complete strategy for how various sub-trees are to be VISITed during the entire course of

[5]This situation would be different for either Warren's coroutine evaluator or the Cohen-Harry evaluator since they could take advantage of fortuitously available attribute-instances.

ıttribute evaluation.

\n optimal sub-protocol-evaluator will likely be one with fewer VISIT-procedures, rather than one built rom a smaller collection of protocols. Thus the heuristics of the protocol propagation algorithm should ›e chosen so as to minimize the number of distinct VISIT-procedures, rather than to minimize the number ·f protocols. All three of the heurstics given in section do this

0. Conclusions

Ve have introduced a terminology and notation, ***semantically-trivial exact covers***, for describing a certain ass of attribute grammar transformations. The definition of this general class of transformations seems ıstified since other researchers have also studied members of this class [15, 5]. Two of these ·ansformations, p-split and c-split, were discussed in detail and it was shown that they preserve some ·operties of the underlying CFGs (e.g. lack of ambiguity) but not others (LR(k)-ness for p-split, LL(k)- ›ss for c-split). The combination, u-split = p-split $\circ$ c-split, transforms any well-defined attribute ·ammar to a uniform attribute grammar, and in the process the transformation finds the corresponding raight-line evaluator for this uniform attribute grammar.

'hen u-split is the identity transformation on G it produces a straight-line evaluator for G. We compared is evaluator with the *ordered evaluators* of Kastens [14] and showed that u-split could be viewed as a ore effective extension of Kastens' strategy.

u-split is not the identity transformation on G then it produces a straight-line evaluator for a larger tribute grammar. We showed how this evaluator can be modified to use the semantic tree of the :ginal G, thus avoiding some potential parsing problems that u-split might have introduced. The ;ulting evaluators are called protocol-evaluators. They can be viewed as a "pre-compiled", optimized :sion of Nielson's *direct evaluators*.

ıally, we compared the protocol-evaluator with the Kennedy-Warren evaluator and concluded that the ›tocol-evaluator used a more goal-directed, top-down strategy for building evaluators, but that in reme cases the Kennedy-Warren evaluator could build smaller evaluators. The sub-protocol-evaluator nbines the best traits of the protocol-evaluator and the Kennedy-Warren evaluator.

e algorithms for building sub-protocol-evaluators (and protocol-evaluators) still contain substantial ounts of non-deterministic choice. Further heuristics could be devised for the protocol propagation ;orithm, either to build smaller evaluators, or to enhance certain other properties of the evaluator. An portant example of such properties would be the ability of the generated evaluators to share storage for ·tain attributes (e.g. attribute stacks [23], "static subsumption" [8]; cf. [24]).

date no experimental research has been done on the efficacy of protocol propagation for the attribute :mmars that people commonly write. Since Kastens' ordering algorithm is a special case of protocol ›agation, Kastens opinion of the wide applicability of ordered attribute grammars [14] suggests that attribute grammars written to describe programming language should present no problems for ocol propagation. Experimental evidence for this is needed, though, and exprience may suggest even er heuristics to use in protocol propagation.

e research is also needed into effective application of the c-split transformation or variants thereof. the purpose of building attribute evaluators we would like to find variants of c-split that *lift* non- ›lutely non-circular grammars to relatively small absolutely non-circular grammars that need not be ›le. One strategy might be to develop heuristics for deciding when new characteristic graphs should be ›d to a non-terminal's set of characteristic graphs; such heuristics would be analogous to those used in ocol propagation.

References

[1] Alfred V. Aho and Jeffery D. Ullman.
The Theory of Parsing, Translation, and Compiling.
Prentice-Hall, 1973.

[2] G.V. Bochmann.
Semantic evaluation from left to right.
Communications of the ACM 19, 1976.
pp. 55-62.

[3] R. Cohen and E. Harry.
Automatic generation of near-optimal linear-time translators for non-circular attribute grammars.
In *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages.*
ACM, January, 1979.

[4] B. Lorho.
Semantic attribute processing in the system DELTA.
In A. Ershov and C.H.A. Koster (editor), *Methods of Algorithmic Language Implementation.*
Springer-Verlag, Berlin-Heidelberg-New York, 1977.

[5] Engelfriet, Joost and Gilberto Filè
Simple Multi-Visit Attribute Grammars.
Technical Report, Department of Applied Mathematics, Twente University of Technology, August, 1980.

[6] I. Fang.
FOLDS, a declarative formal language definition system.
Technical Report STAN-CS-72-329, Stanford University, 1972.

[7] R. Farrow.
Experience with an attribute grammar based compiler.
In *Conference Record of the Ninth ACM Symposium on Principles of Programming Languages.*
ACM, January, 1982.

[8] Rodney Farrow.
LINGUIST-86 Yet another translator writing system based on attribute grammars.
In *Proceedings of the SIGPLAN 82 Symposium on Compiler Construction.* ACM, June, 1982.

[9] H. Ganzinger, K. Ripken, and R. Wilhelm.
Automatic generation of optimizing, multipass compilers.
In *Proceddings of IFIP 1977.* 1977.

[10] M. Jazayeri and K.G. Walter.
Alternating semantic evaluator.
In *Proceedings of ACM 1975 Annual Conference.* ACM, 1975.

[11] M. Jazayeri, W.F. Ogden, and W.C. Rounds.
The intrinsically exponential complexity of the circularity problem for attribute grammars.
Communications of the ACM 18, 1975.

[12] Diane Pozefsky and Mehdi Jazayeri.
Attribute evaluation without a parse tree.
Technical Report, University of North Carolina, Chapel Hill, 1979.

[13] Uwe Kastens, Brigitte Hutt, and Erich Zimmermann.
GAG:A Practical Compiler Generator.
Spring-Verlag, Berlin-Heidelberg-New York, 1982.

[14] U. Kastens.
Ordered attribute grammars.
Acta Informatica 13, 1980.

[15] T. Katayama.
Translation of attribute grammar into procedures.
Technical Report CS–K8001, Dept. of Computer Science, Tokyo Institue of Technology, July, 1980.

[16] K. Kennedy and S. K. Warren.
Automatic generation of efficient evaluators for attribute grammars.
In *Conference Record of the Third ACM symposium on Principles of Programming Languages.* ACM, 1976.

[17] K. Kennedy and J. Ramanathan.
A deterministic attribute grammar evaluator based on dynamic sequencing.
ACM TOPLAS 1, 1979.

[18] D. E. Knuth.
Semantics of context-free languages.
Mathematical Systems Theory 2, 1968.
correction in volume 5, number 1.

[19] H.R.Nielson.
Computation sequences: A way to characterize subclasses of attribute grammars.
Technical Report, Aarhus University, Denmark, 1981.

[20] H.R.Nielson.
Using computation sequences to define attribute evaluators.
Technical Report, Aarhus University, Denmark, 1981.

[21] D. Pozefsky and M. Jazayeri.
Space efficient storage management in an attribute evaluator.
ACM TOPLAS 3(4), 1981.

[22] Kari-Jouko Raiha, M. Saarinen, E. Soisalon-Soininen and M. Tienari.
The Compiler Writing System HLP (Helsinki Language Processor).
Technical Report A-1978-2, Dept. of Computer Science, Univ. of Helsinki, 1978.

[23] M. Saarinen.
On constructing efficient evaluators for attribute grammars.
In C. Ausiello and C. Bohm (editor), *Automata, Languages, and Programming: 5th Colloquium.* Springer-Verlag, Springer-Verlag, New York, 1978.

[24] Ravi Sethi and Jean-Claude Raoult.
Storage sharing during attibute evaluation.
Feb, 1983.
Bell Labs technical report, in preparation.

[25] S. K. Warren.
The coroutine model of attribute grammar evaluation.
PhD thesis, Rice University, May, 1976.