SPINE
A Synthesizer for Practical Incremental Evaluators

Kannan Muthukkaruppan
Computer Science Division
University of California,
Berkeley, CA 94720.

May 19, 1994

Abstract

SPINE is a system for efficiently generating practical incremental evaluators based on recursive procedures for the *strongly non-circular* class of attribute grammars (AGs). Several interactive language-based software development environments use incremental evaluation of attribute grammars for context-sensitive static semantic analysis. The key advantages this system offers over other existing incremental AG systems are ease of evaluator construction, effective consumption of space, applicability to a large class of AGs, ability to handle multiple site attribute tree transformations and close to *optimal* performance. ASPEC is the AG-description language that has been specially designed for the SPINE system.

SPINE has been used innovatively in the Ensemble software development environment to provide advanced incremental formatting of documents. ASPEC serves as the format specification tool for Ensemble documents. The ASPEC language provides several powerful default mechanisms which make these specifications very concise.

1 Introduction

Attribute grammars (AGs) [Knu68] are a formal and practical method for rule-based specifications of computations on tree structures. Attribute evaluation is a process by which the values of attribute instances on the nodes of the tree are computed (tree decoration). Several applications, like compiler optimizations and syntax-directed editing, can be described by tree transformation operations. These operations could, in general, invalidate several attribute instances in the tree. Incremental attribute evaluators attempt to restrict the amount of recomputation required to achieve a valid attribution of the tree again.

One of the primary goals in development of language-based programming environments is to achieve good incremental interactive performance for the variety of services they provide. These environments are typically used to edit, analyze (both syntax and semantics), execute and debug programs. Such systems, (like Ensemble [Gra92], [Mun92] and the Cornell Program Synthesizer [Rep83]), allow creation of language-based environments for different languages from formal specifications of the syntax, semantics and presentation[1].

Attribute grammars, being powerful specification tools for context-free grammars, provide a good framework for specifying and generating language based editors. The static semantic information of programs being edited by these systems is represented as an attribution of the program syntax tree.

In such language-based environments the semantic information is used for several purposes such as giving the user feedback on compile-time errors, code generation, debugging support, context-sensitive editing and browsing. Tree modifications, as a result of editing operations, invalidate attribute instances. Invalidated attribute instances need to be restored to their correct consistent values. High demands on interactive performance of these systems necessitates the presence of efficient incremental attribute evaluation algorithms for attribute grammars.

The goal of this work has been to design and implement a simple yet practical system for generating efficient incremental attribute evaluators for the *strongly non-circular* AGs (SNC) [CF82] class of attribute grammars. This category of AGs is equivalent to the *absolutely non-circular* (ANC) class defined in [KW76]. The system is called **SPINE**, a **S**ynthesizer for **P**ractical **In**cremental **E**valuators. The construction of the evaluator, given an AG description, is fairly straightforward; it primarily involves the SNC membership test. SPINE generated evaluators are space efficient and offer close to *optimal* performance (Section 4.2.6). SPINE also allows editing operations at multiple subtrees before initiating an incremental reevaluation. **ASPEC**, is the AG-description language that has been specially designed for SPINE system (Section 5).

SPINE has been employed in a very innovative and practical application: incremental formatting of documents in Ensemble [Gra92], an interactive software development environment (Section 6). The SNC class of attribute grammars seems to be quite sufficient to express a wide variety of complex presentations for documents.

2 Related Work

Attribute grammars offer nice language specification features; they are declarative in nature (non-procedural), exhibit structure and have locality of reference. The key property that makes them

[1]In this report the word *presentation* is used in the Ensemble context to refer to formatting information for the syntactic entities of documents such as layout, font name, font size, indentation, color etc.

useful, however, is that they are *executable*, meaning that it is possible to automatically construct evaluators for them. For the unrestricted class of AGs the attribute evaluation procedure is in general non-deterministic. Most evaluator construction algorithms restrict the class of attribute grammars they accept in order to improve the compute time and memory usage of the generated attribute evaluator. Restricting the class of attribute grammars also reduces the complexity of the constructor itself. For a good survey on the basic attribute evaluation techniques refer to [Eng84].

There has been a lot of interest in construction of attribute evaluators that are *incremental*, especially in the context of incremental programming systems. An *incremental* evaluator is one that does not naively recompute the values of all attribute instances whenever a tree transformation occurs, but rather, intelligently restricts the amount of recomputation required to make the attribution of the tree consistent again by using information from the previous computation. By definition, an *optimal* incremental attribute evaluator runs in $O(|AFFECTED|)$ time [Rep83], where $AFFECTED$ is the subset of attribute instances of the tree that require new values as a result of a modification to the derivation tree. Observe that the $AFFECTED$ set is a function of the tree transformation itself, and hence the process of restoring tree consistency must include identification of this set of attributes by the evaluator.

Two naive approaches to incremental attribute evaluators are *change propagation* and *nullify/reevaluate*. In the change propagation approach the changes of attribute values are propagated throughout a fully attributed tree starting at the site of the transformation by following attribute dependencies. A work list of possibly inconsistent attributes, that must be reevaluated because one of their arguments changed, is maintained. As is pointed out in [Rep83], this approach can result in exponential behavior because it is sensitive to the order in which the work list is manipulated. The nullify/reevaluate style of evaluation involves two tree traversal phases: the nullify phase where a *null* value is propagated to all attributes that depend on the change site (directly or by transitivity); and the reevaluate phase which is used to propagate consistent new values to the nullified attributes. This algorithm is linear in the number of attributes considered. It does not however fall in the category of *optimal* evaluators because the set of attributes considered here is the set of all attributes that depend on the modification site, which is a superset of attributes that actually require new values (given by the $AFFECTED$ set).

In the Synthesizer Generator System [Rep89], optimal incremental evaluators have been developed for two classes of AGs, the ordered-AG (OAG) class [Kas80] and arbitrary non-circular class. The evaluators for the arbitrary non-circular class are scheduled dynamically and are highly storage intensive because of the amount of evaluator state they are required to keep around at run-time. The evaluators for the OAG class perform incremental updating by a Visit-Sequence driven change propagation algorithm (described in [Rep89]). Visit-Sequences are essentially sequences of tree walk operations and computations that determine the evaluation order for attributes of a tree. (The idea of Visit-Sequences was introduced in [Kas80].) There are two main problems with this approach:

- Construction of Visit-Sequence evaluators is inherently non-trivial. A significant and complex amount of analysis is involved in determining visit-sequences, and then partitioning them to obtain incremental evaluation plans.

- A more fundamental problem with the change-propagation style of incremental algorithms is that change-propagation can be initiated only from a single (editing) site in the derivation tree. In several language-based editing environments (including Ensemble, [Mun92]), editing operations trigger incremental reparse that can result in subtree replacements at multiple sites. The single-site editing requirement is therefore unnecessarily restrictive.

The FNC-2 system [Jou91] implements incremental attribute evaluators for *doubly non-circular* (DNC) class of attribute grammars, which are a strict subset of the SNC class. The advantage of this evaluator is that reevaluation can begin at any node of the tree because of a certain property of the DNC class, namely their *argument selectors* are closed "from below" as well as "from above" [Fil87]. The approach here is to first generate an exhaustive evaluator and later add to them a set of "semantic control" functions (based on comparisons to old values of attribute instances) that limit reevaluation process to affected instances.

Vogt, et. al. [Vog91] have proposed incremental evaluators based on *visit-function* caching. Their scheme allows multiple site tree transformations as well. Provision for multiple site editing involves a coordination overhead between the subtrees that undergo transformation. This results in sub-optimal behavior; the overhead usually involves invalidation of all attributes on the paths to the root from all the editing sites. The time characteristic of this class of algorithms is therefore $O(|AFFECTED| + |attrs\_on\_paths\_to\_root|)$. Their evaluators are, however, restricted to the ordered-AG (OAG) class of attribute grammars. The SPINE approach is similar in flavor to this style of evaluator but works for the far richer SNC class of attribute grammars.

Peckham [Pec90] achieved a time bound of $O(|AFFECTED| \cdot \log n)$ for evaluators that handled multiple subtree replacements, where k is the number of subtree replacements and n is the total size of the tree. Again, this algorithm works only for a specialized subclass of OAGs and also exhibits large memory usage characteristics for the data structures required.

3 The SPINE approach

The SPINE system produces incremental evaluators for the **Strongly Non-Circular** (SNC) class of attribute grammars. The SNC class is the largest class of attribute grammars for which efficient evaluators can be constructed and membership can be determined in polynomial time. In practice it has been observed that the SNC class is general enough to express almost any useful attribute grammar. Section 4.2.1 presents an informal characterization of the SNC attribute grammars (also called the Absolutely Non-Circular class). This view has been described elegantly in [Jou84] and is based on the theoretical construction by B. Courcelle and P. Franchi-Zannettacci [CF82].

The non-incremental method of attribute evaluation originally proposed by Courcelle and Franchi-Zannettacci [CF82] and implemented by Jourdan [Jou84] forms the theoretical basis for the SPINE implementation. This approach, which is often called the "Synth-Function Approach", corresponds to recursive procedure schemes for attribute evaluation. SPINE uses *function caching* to incrementalize the recursive evaluation process. This is similar to the technique used in the Colander II system ([Mad93]) currently under development at Berkeley.

In the "Synth-Function" approach, each synth-function is a procedure that computes the value of a synthesized attribute of a certain phylum. The arguments to the function are: a pointer to the subtree issuing at the node corresponding to the desired attribute instance and the set of inherited attributes upon which the computation of the synthesized attribute might depend. The basic idea in synth-function caching is that the previous result of a synth-function call is cached. If the set of arguments to a synth-function exactly matches its previous invocation, then the old value of the corresponding synthesized attribute is simply returned without recomputation. This potentially avoids several other attribute computations. Since root nodes do not have any inherited attributes, reevaluation in such evaluators is done by invoking "synth-functions" of synthesized attributes at the root node. Naively implemented, this evaluator simply returns the values of synthesized attributes at the root node of any tree structure. Suitable modifications are needed to make sure that all attribute instances, both inherited and synthesized, get computed and stored at the appropriate tree nodes. The attribute evaluation technique with all the required modifications is discussed in detail in Section 4.2.

The "synth-function" evaluators overcome the aforementioned two problems about visit-sequence based evaluators. The following are the merits of using the "synth-function" approach:

- It allows for multiple editing sites, because it is not based on a change-propagation scheme. Coordination between sites is achieved by simply invalidating all attributes along paths to the root from the various editing sites and then triggering a reevaluation at the root. The ability to handle multiple subtree replacements is a useful feature, especially in language based environments that have support for incremental parsing. Often a single edit operation can trigger an incremental reparse that modifies the abstract syntax tree at multiple sites.

- A much less complex analysis is required for evaluator construction. The analysis basically involves the SNC membership test, which is polynomial time.

- The code generated by these evaluators closely resembles the input AG description, and is therefore easy to read independent of the input description. This greatly helps debugging of AG descriptions in case of errors.

- The "synth-function" approach has an inherent advantage in terms of balancing a space performance tradeoff. The primary performance benefit in the evaluator comes at the expense of space, due to the synth-function caching. Observe that caching is not a prerequisite for correctness of the evaluator, but rather just a performancing enhancing technique. Therefore, the algorithm provides the flexibility of turning off/on caching (storage of results of previous computations) selectively for certain set of attributes, thus allowing a finer control over trading time for space or vice-versa.

The primary disadvantages of the "synth-function" style of evaluators are as follows. (Some of these issues are disussed in greater detail in later sections.)

- They are only close to, but not really *optimal*.

- Attribute evaluation in this approach involves separate tree traversals for each synthesized attribute (in some sense more pointer chasing), unlike visit-sequence based approaches, which attempt to minimize the number of visits to tree nodes.

- It is possible that an inherited attribute is evaluated multiple times during a certain pass of attribute reevaluation, e.g., when two synthesized attributes depend on an inherited attribute, then each of their synth-functions needs to be passed the value of the inherited attribute on which they both depend.

4 Implementation of SPINE

First an overview of the SPINE system is presented in Section 4.1. Then the technique used for incremental attribute evaluation in SPINE is described in Section 4.2. Section 4.3 details the interface that client programs using the evaluator (PINE) need to implement.

4.1 System Overview

The SPINE system generates incremental attribute evaluators for AG descriptions specified in the ASPEC attribute definition language (Section 5). It compiles an ASPEC input description into a hard-coded C++ evaluator called **PINE** (a **P**ractical **I**ncremental **E**valuator).

The generation and evaluation time interaction between the various modules of the system is shown in Figure 1.

The front-end of SPINE is an ASPEC scanner and parser. It does static-semantic analysis of the input to ensure that the input description conforms to ASPEC requirements. (See ASPEC error detection in Section 5.10). SPINE then performs automatic generation of missing rules (Section 5.4) by using the phylum default and production default sections of ASPEC (Section 5.5, 5.6). It performs analysis to determine the type information of the attributes, parses the right hand side of semantic rules of the AG into abstract derivation trees, and computes dependency information between attributes. As a result of all the analysis, an Abstract Attribute Grammar is obtained.

The Abstract Attribute Grammar is then tested for membership in the Strongly Non-Circular class of AGs. If the test fails, SPINE reports the cause for circularity and aborts. The analysis done during the SNC membership test along with the Abstract Attribute Grammar is used to produce PINE code.

A client (tree transforming application) that wishes to use PINE code for incremental attribute evaluation is required to implement the PINE client interface (Section 4.3).

4.2 Incremental Attribute Evaluation: the technique

Jourdan ([Jou84]) proposed evaluators based on recursive procedures for the SNC class of attribute grammars. The construction is based on the following very important result proved in [CF82],

For any AG the value of a synthesized attribute at a node of a derivation tree depends only on:
a) the subtree issued from that node.
b) the value of the inherited attributes at that node.

4.2.1 Informal characterization of SNC attribute grammars

For each synthesized attribute occurrence we would like to find the set of inherited attributes upon which it depends. This set however depends not only on the phylum (abstract non-terminals) in which the attribute occurs but also on the context of that phylum in the derivation tree. The term *argument selector* is used to refer to the function that associates with any synthesized attribute the set of inherited attributes which will be passed as arguments to the function computing that attribute. The problem is that the determination of the argument selector is dynamic because it is dependent on the parse tree. We must therefore find an approximate solution for the argument selector that does not specifically depend on the parse tree. This requires us to be pessimistic about the set of dependencies that can exist in any parse tree.

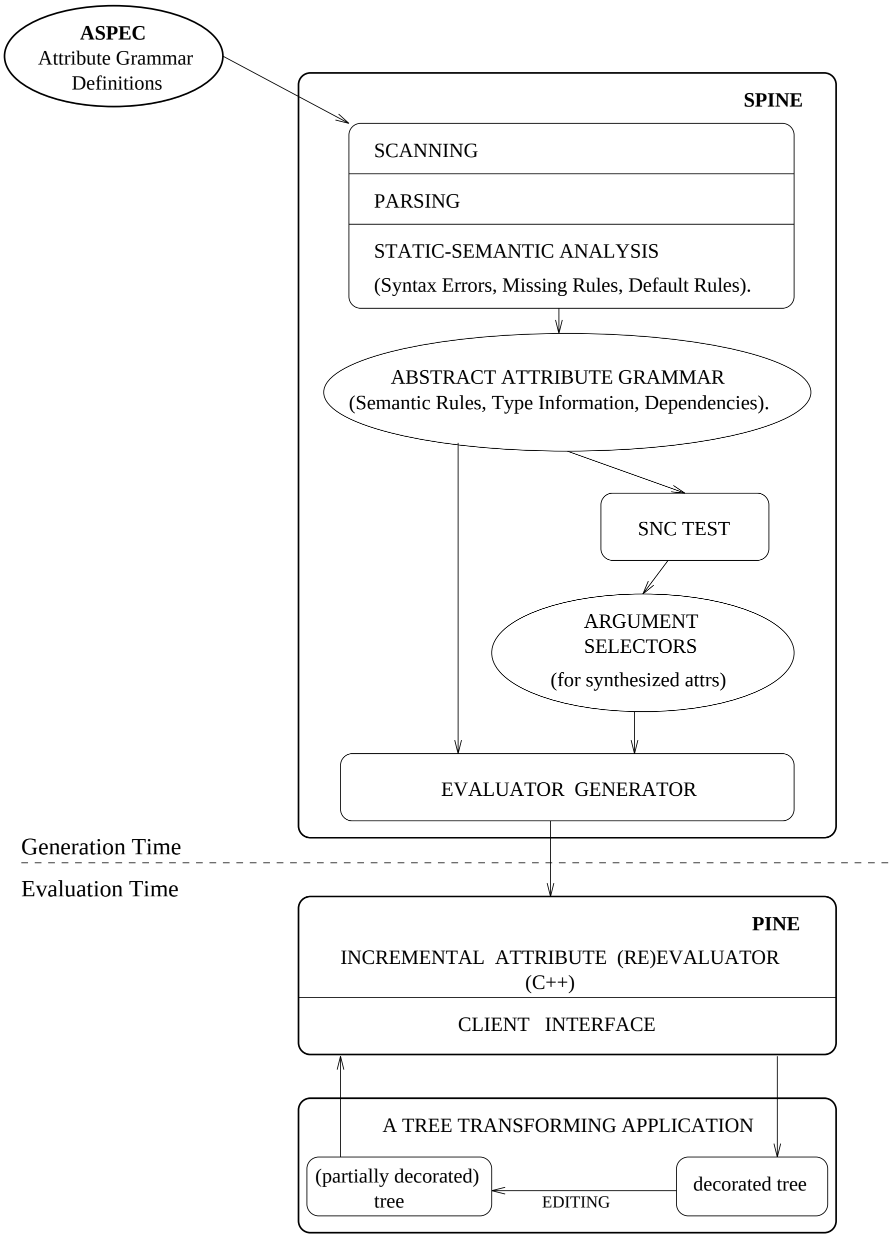

Figure 1: SPINE system architecture

For the evaluator construction to be valid the argument selector must satisfy two important conditions:

- it must be *closed*: The argument selector of a synthesized attribute must at least contain all the inherited attributes on which it might depend in any tree.

- it should be *non-circular*, i.e., an inherited attribute in the argument selector of a synthesized attribute cannot itself be dependent on the synthesized attribute.

The Strongly Non-Circular attribute grammars are a class of attribute grammars for which it is possible to *statically* determine for each synthesized attribute a *minimal* set of inherited attributes upon which its computation *may* depend in *any* derivation tree. This minimal set of inherited attributes is referred to as the *minimal closed argument selector*, often abbreviated as simply the *argument selector* of the synthesized attribute. Furthermore, the argument selectors are guaranteed to be non-circular for SNC AGs. Although SNC class is a proper subset of the class of non-circular attribute grammars, it covers almost all practical applications.

The algorithm to test membership in the SNC class involves as the first step determination of the minimal closed argument selector function. We then check to see if the minimal closed argument selector is non-circular, in which case the attribute grammar belongs to the SNC class. The algorithm is presented in [Jou84]. It is polynomial in time and space.

4.2.2 Ordinary Synth-Functions

For each synthesized attribute we can therefore write a function that takes as arguments some set of inherited attributes (its argument selector to be precise) and a pointer to a subtree. It returns the value of the synthesized attribute at the root of the subtree. These are the "Synth-Functions" used by Jourdan's recursive (non-incremental) evaluators.

Let `s` be a synthesized attribute of phylum `X` and the set {`a1, ...,aN`} its argument selector. The ordinary version of the synth-function `_s_X` then looks like:

```
type_s   _s_X(node *subtree, type_a1 a1, ..., type_aN aN) {
    type_s  s;
    switch (subtree->prodNum()) {
    ...
    case <p>:

          s = <Semantic Rule for 's' in <p> with attribute
               occurrences in the rule substituted by their
               attribute definitions>;
          break;
    ...
    }
    return s;
}
```

4.2.3 Memoizing Synth-Functions

The key observation in generating incremental evaluators is that a "Synth-Function" must return the same value as on the previous invocation if the values of the parameters do not change. When there is no stored value of the attribute at the root node of the subtree (either because it is being computed for the first time or because it was invalidated due to an editing operation), we need to recompute the attribute. If a stored value does exist and the input arguments match those of the previous invocation we can simply return the stored value of the attribute. Otherwise, we need to recompute the value of the attribute.

Whenever the value of synthesized attribute is recomputed, we memoize the result. This involves storing the values of both the attribute and its argument selectors in the tree node.

Let `s` be a synthesized attribute of a phylum `X` and the set {`a1, ...,aN`} its argument selector. The memoized version of the Synth-Function `_s_X` then looks like:

```
type_s   _s_X(node *subtree, type_a1 a1, ..., type_aN aN) {
    type_s  s;

    if ((stored value of 's' at subtree exists) &&
        ((a1, ..., aN) == (memoized values of previous call))) {

         /* no need to recompute 's' */
         return stored value of 's';

    }
    switch (subtree->prodNum()) {
    ...
    case <p>:
          s = <Semantic Rule for 's' in <p> with attribute
               occurrences in the rule substituted by their
               attribute definitions>;
          break;
    ...
    }

    Memoize; /* Save a1, ..., aN in subtree node */
    Store s in subtree node;
    return s;
}
```

4.2.4 The "_update" Attribute

In the memoizing scheme, whenever a synth-function is invoked, the value of the corresponding synthesized attribute gets stored in the root node of the subtree that is passed to the synth-function as argument. But how do we ensure that the synth-functions of all attributes get invoked at every node of the tree? Also, how do all the inherited attributes at the nodes get computed and stored?

One convenient way to ensure that the above two requirements are met is to create a dummy synthesized attribute, **_update**, for each phylum. This attribute is important because of the side-effects caused by its synth-function. It is forced to depend on all the inherited attributes of the phylum. Therefore the argument selector of **_update** attribute of a phylum is all the inherited attributes at the phylum. This ensures that in the body of the synth-function for **_update** all the inherited attributes of its phylum will be available to be stored. Also the body of this synth-function invokes synth-functions of all other synthesized attributes at the node, then continues tree traversal by invoking the synth-functions corresponding to the **_update** attribute of each of its children.

Let X be a phylum, with s1, ...sK as its synthesized attributes (excluding the _update attribute), and let a1, ..., aN be its inherited attributes. Then the synth-function for this _update attribute has the following form:

```
void   _update_X(node *subtree, type_a1 a1, ..., type_aN aN) {

    if (( '_update' is defined at root node of subtree) &&
        ((a1, ..., aN) == (memoized values of previous call))) {

         /* No inherited attribute of this subtree changed */
         /* Hence no need to traverse this subtree */
         return;
    }

    /* Compute all the synth attrs of X */
    Invoke Synth-Functions for s1, ..., sK;

    switch (subtree->prodNum()) {
    ...
    case <p>:
        Invoke _update synth functions for each child of operator <p>;
    ...
    }

    Memoize; /* save all the inherited attributes (a1, ..., aN) at X */
    Set defined status of _update attribute to TRUE;
}
```

The additional dependencies induced as a result of the **_update** attribute cannot convert an original SNC AG to a non-SNC AG. The SPINE system automatically generates **_update** attributes for all phyla with the required dependencies.

4.2.5 Initiating a Re-Evaluation

After a sequence of tree editing operations have been performed, in order to initiate an attribute re-evaluation, all attributes along paths from the editing sites to the root of the tree are invalidated.

Then, we simply invoke the Synth-Function for the **_update** attribute of the **rootPhylum**, by passing it a pointer to the root of the tree. No other arguments need to be passed to this function because the root phylum can have no inherited attributes.

4.2.6 Storage Requirements and Time Analysis

The evaluator has a time characteristic of $O(|AFFECTED| + |attrs\_on\_paths\_to\_root|)$. The term "$attrs\_on\_paths\_to\_root$" appears due to the fact that for each editing site we clear all the attributes on the path from the editing site to the root. This implies that our algorithm does not fall in the class of *optimal* incremental evaluators. In fact, in the worst case, i.e., when the tree is degenerate (tall and thin) or when the number of editing sites is large, the entire set of attributes over the tree may have to be recomputed. Such cases arise very rarely and hence are not of serious concern.

Observe that since function caching applies only to the synthesized attributes, the value of an inherited attribute might be computed multiple times in a single pass of attribute reevaluation. Inherited attributes in most applications often serve as simple transfer attributes that propagate values down the tree by copy-rules. Usually no expensive computation is involved in determining the value of an inherited attribute. Hence, occasionally computing an inherited attribute multiple times is not costly.

Apart from the storage required to store the values of all attribute occurrences in the nodes of the derivation tree, for each synthesized attribute of a phylum (node), storage is required to memoize values of its argument selector. This storage is part of the class definition generated by PINE for this phylum. For each synthesized attribute occurrence a *defined* bit is also maintained in the corresponding phylum instance. This indicates if there is a stored value for the attribute in this phylum.

4.3 The PINE-Client Interface

The PINE-Client interface is object-oriented and is implemented in C++. In the PINE world, the tree nodes correspond to the class **AgNode**. The class has the following interface. Please note that this interface has been implemented differently (Section 6.5) for Ensemble because of the nature of the requirements of the Ensemble system.

```
class AgNode {
  public :

    AgNode(char *phylum);
    _PhylumData *attrData;

    //---- These need to be implemented by the client. ----

    virtual int prodNum() = 0;
    virtual AgNode *parent() = 0;
    virtual AgNode *child(int n) = 0;
    virtual void setVisited() = 0;

    //---- These are implemented by PINE  ----
    void subtreeChange();
    void reEval();
}
```

The tree nodes of the client application inherit from the AgNode class. PINE, during evaluation, needs to know information such as the operator (production) that applies at a node (prodNum()), the parent node (parent()) and a specific child of a node (child(num)). These are C++ pure virtual functions in the AgNode class, and are implemented by the client. [The reason for making these functions virtual is that the AgNode class does not maintain any navigational information (like pointer to its parent node or children). These are implemented by the client in any manner it chooses to do so.] The setVisited() function is invoked by PINE whenever a node is visited by it during incremental attribute re-evaluation. The client can chose to implement setVisited() however it wants. Typically, this can be used by the client to study the performance gain of using an incremental algorithm.

The methods subtreeChange() and reEval() in the AgNode class are implemented in PINE. Whenever a subtree is modified, the client should invoke the subtreeChange() method at the root of the modified subtree. Subtree changes could be invoked by the client at several editing sites (in case there are multiple subtree replacements). Then to trigger attribute reevaluation, the client simply invokes the reEval() function to obtain a fully decorated tree again.

Last but not the least, the client needs to be able to access the values of the attributes computed at a particular AgNode. PINE code contains class definitions for each of the non-terminals (phyla) in the AG. These are generated automatically by SPINE. The class definition for a particular phylum contains storage for the attributes of that phylum. All phylum class definitions inherit from a base class called _PhylumData. PINE provides the implementation for the _PhylumData. The member attrData of any AgNode instance is a pointer to a derived class of _PhylumData. The derived class it points to depends on the phylum that corresponds to this instance of the AgNode.

For example, let P be a phylum with 2 attributes attr1 and attr2 of type int and bool respectively. Corresponding to this phylum P, as we mentioned before, PINE generates the following class definition.

```
class P : public _PhylumData {

  public:
    int attr1;
    bool attr2;

    ... /* other book-keeping stuff */
}
```

To access, say attribute attr2 of phylum P at AgNode node, simply cast the attrData member of AgNode into a pointer to class P, and then access the desired field from the class.

```
((P *)(node->attrData))->attr2
```

5 ASPEC as a AG-description language

ASPEC is a new, specialized language that has been designed for specifying attribute grammar descriptions for the SPINE system. ASPEC is an applicative programming language (which means there are no assignable variables or side-effects, just pure expressions and functions). This is useful for programming safety and reliability. The language has natural semantics that make it easy to read, learn and use. ASPEC specifications of AGs are completely independent of the evaluator (e.g. incremental, exhaustive) that one may want to generate for it.

The syntactic base of AGs written in ASPEC is not required to be a concrete grammar. The AGs in ASPEC are structured by an *abstract syntax*. This enables us to have a "semantic view" of the application for which an AG is being written, rather than an enforced syntactic view. The abstract syntax must be input as a part of the ASPEC description. Using an abstract syntax relieves the ASPEC programmer from cumbersome syntactic constraints that might arise from a particular parsing technique and also enables maintenance of smaller sized derivation trees.

ASPEC provides for heterogeneous, fixed arity *operators* (abstract productions). Each *phylum* (abstract non-terminals or node) has set of attribute declarations that corresponds to the set of attribute instances that will need to be computed to decorate tree nodes of its type.

The inherited attributes of an operator and the synthesized attributes of its children are called the *input* attributes of the operator. The synthesized attributes of an operator and the inherited attributes of its children are called the *output* attributes of the operator. For each output attribute of an operator, there must exist a *semantic rule* (attribute definition) in the ASPEC input.

For the complete ASPEC grammar refer to Appendix A. A sample ASPEC input description is given in Appendix B. Next, several useful and interesting language features provided by ASPEC are discussed.

5.1 No Normal-Form restriction on Semantic Rules

Several attribute grammar systems require their AG-descriptions to be in *normal* form. The normal form restriction implies that the right hand side of a semantic rule for an *output* attribute of an operator can have references only to *input* attributes of the operator. In ASPEC we relax this constraint by allowing *non normal* form AGs in which the right hand sides of the semantic rules can have both *input* and *output* attribute occurrences. This relieves the burden of the ASPEC programmer having to know about the technical details of AGs.

For example, consider the following production and the semantic definitions for its attributes.

```
S --> A B
{
  $$.Width = $1.Width+$2.Width;
  $1.Left = $$.Left+5;
  $2.Left = $1.Left+$1.Width+2;
}
```

In the above operator, the attribute instances `S.Width, A.Left` and `B.Left` are the output attributes; `S.Left, A.Width` and `B.Width` are the input attributes. The semantic rule for `$2.Left`, i.e., `B.Left` has a reference to another output attribute `$1.Left` i.e. `A.Left`. Such an attribute reference is a non formal form attribute reference.

The SPINE system's SNC membership test and evaluator construction work when applied to non normal form AGs. Non normal forms do not increase the expressive power of the AG, but simply allow us to write more compact AGs. A non normal form AG can be converted into an equivalent normal form AG by a simple textual replacement of the attribute occurrences in the semantic rule that are in non normal form by the right hand side of the semantic rules that define them.

In fact, for non normal form occurrences of synthesized attributes, SPINE simply generates a call to the corresponding "Synth-Function", rather than expand its definition. In the example above, for the non normal form occurrence of `$1.Left` in the semantic rule for `$2.Left` we can generate a call to the "synth-function" for `Left` attribute of phylum `A` rather than textually substituting the semantic rule for `$1.Left` which is `$$.Left+5`. Firstly, this reduces the size of evaluator code. Secondly, it improves the evaluator's run-time performance due to the possibility of a cache hit in the memoized synth-function, if that attribute has already been computed.

5.2 Extern Function Calls

The right hand side of the semantic rules can have function call invocations. These functions are assumed to be defined external to the AG description.

For example,

```
P --> A B C
{
  $$.a = foo1($1.a, foo2($3.c)) + 3;
}
```

In the above example the right hand side of the semantic rule has references to two external functions `foo1` and `foo2`. These functions are implemented external to PINE. The generated PINE code contains a reference (`#include`) to a special header file. The exact name of this header file depends on the name of the ASPEC file. Say, "hello.aspec" is the name of the ASPEC input file. The generated PINE code `#include`s "hello-helper.h" file name. This file can contain the `extern` declarations for all the extern function calls in the ASPEC input. Typically, the required functions are implemented in "hello-helper.cc" file.

5.3 Aggregate Function Calls

Aggregate function calls can occur in the right hand side of the semantic rules. The syntax for aggregate function calls is '@<funcname>.<attr>'. In the context of an operator with n children, this aggregate function call is a compact way of writing '<funcname>(n, $1.<attr>, ..., $n.<attr>)'.

This is best illustrated by an example.

```
P --> A B C
{
  $$.width = @Max.Right - @Min.Left;
}
```

The right hand side of the above semantic rule is equivalent to the expression:

```
Max(3, $1.Right, $2.Right, $3.Right) - Min(3, $1.Left, $2.Left, $3.Left)
```

Here, `Max` and `Min` are assumed to be externally defined variable argument functions. Hence the number of children is passed as the first argument to each one of them. Again, the extern declarations of these variable argument functions must also be placed in the special '.h' file as in the case for external function calls (Section 5.2). Aggregate function calls are very useful in writing compact AGs as they lend themselves well to writing default semantic rules that apply to several different productions. See Section 5.6 for an example.

5.4 Automatic Generation of Missing Rules

ASPEC provides a very powerful mechanism for automatically generating missing default rules. When used wisely, it lends to very compact AG descriptions. It is worth mentioning that the design of this component was primarily motivated by the desire to keep AG descriptions for layout attributes of Ensemble documents small and simple to write.

When an explicitly specified semantic rule is missing for an *output* attribute of an operator we attempt to use a semantic rule from the *phylum defaults section* for this attribute. If we are unable to do so, we consult the *production defaults section* for an applicable semantic rule. If we find none, we declare an error. Next we discuss ways to use phylum and production default sections for automatically generating missing rules.

5.5 Phylum Defaults Section

When an output attribute occurrence, say attribute x of phylum P, of an operator does not have a semantic rule explicitly specified, we check to see if a default semantic rule for this occurrence has been defined in the phylum defaults section. If such a *phylum default* rule does exist for attribute x of phylum P, we use it as the definition for x provided it is *legal* in the context of the operator in which it is being applied.

Furthermore, the right hand side of a semantic rule in this section can contain *relative* attribute references (`$/n.attr`), where `n` is an integer.

The following example illustrates interesting ways of using phylum default rules and *relative* attribute references.

```
%% Phylum Defaults Section
ID:
{
  Bold = true;
  Left = $/-1.Left + $/-1.Width;
}
SEMI:
{
  Left = $/-1.Left + $/-1.Width;
}
...
%% Semantic Rules Section
/* operator 1 */
ASSMT --> ID EQ NUM SEMI
{
  ...
}
/* operator 2 */
DECL --> TYPE ID SEMI
{
  ...
}
```

In the above example assume that in both `operator 1` and `operator 2` there are no explicit semantic rules for the `Bold` attribute of `ID` and `Left` attribute of `ID` and `SEMI` (semicolon). The *phylum default* rule for `Bold` attribute of `ID` can be applied in the context of both operators. The `phylum default` rule for `Left` attribute of `SEMI` can also be used for both operators. In `operator 1` this rule translates to `SEMI.Left = NUM.Left + NUM.Width`. In `operator 2` this rule translates to `SEMI.Left = ID.Left + ID.Width`. The `phylum default` rule for `Left` attribute of `ID` can be applied only for `operator 2`. This is because, $/-1 is not legal with respect to `ID` in `operator 1`, since ID has no left sibling in this context.

The use of **$n.attr** notation in semantic rules of ASPEC over the traditional **phylumName.attr** style for attribute references makes it possible to write useful default rules (as shown in the example above) that translate into different rules depending on the context. (The `$n.attr` notation also usually requires fewer key strokes!)

5.6 Production Defaults Section

When we are unable to find a semantic rule for an attribute in the *semantic rule* section or the *phylum default* section, we attempt to find one in the *production default section*.

In this section two symbols `i` and `_n` have special meaning. Let `op` be the operator, with a phylum occurrence `P` that has an attribute `attr` for which we are attempting to find a semantic rule. In this context `_n` refers to the number of children of the operator `op`, and `i` is the index (child number) of this occurrence of `P` in `op`. `i` is used only for phylum on the right hand side of the operator. Hence legal values of `i` range from 1 to `_n`. If `P` is in the left hand side of the operator, only the `$$.<attr>` rules are relevant.

`$i.<attr>` rules in this section could optionally be supplied an additional qualifier that restricts the range of `i` values for which the rule is applicable. For example, a qualifier range `[2, _n]` implies that the rule can be applied for all but the first child of an operator.

For example,

```
%% Production Defaults Section
  /* _n, i are special symbols in this section */
  $$.Width = @Max.Right - @Min.Left;           /* for LHS phylum */
  $i.Bold = false;                      [1, _n] /* for all children */
  $1.Top = $$.Top;                              /* for first child */
  $i.Top = $(i-1).Top + $(i-1).Height;  [2, _n] /* except first child */
...
%% Semantic Rules Section
DECL --> TYPE ID SEMI
{ }
ST_LIST --> ST ST_LIST
{ }
```

In the right hand side of `$i` rules, attribute references can be of the form `$<i-expression>.<attr>`, where `<i-expression>` is an expression involving `i` as a variable.

Assume that the `Semantic Rules Section` in the above ASPEC description does not have any semantic rules for Top, Width and Bold attributes. Then, the following rules are automatically generated:

```
DECL --> TYPE ID SEMI
{
  $$.Width = Max(3, $1.Right, $2.Right, $3.Right)
           - Min(3, $1.Left, $2.Left, $3.Left);
  $1.Bold = false;
  $2.Bold = false;
  $3.Bold = false;
  $1.Top = $$.Top;
  $2.Top = $1.Top + $1.Height;
  $3.Top = $2.Top + $2.Height;
}
ST_LIST --> ST ST_LIST
{
  $$.Width = Max(2, $1.Right, $2.Right) - Min(2, $1.Left, $2.Left);
  $1.Bold = false;
  $2.Bold = false;
  $1.Top = $$.Top;
  $2.Top = $1.Top + $1.Height;
}
```

5.7 Type checking

All attribute declarations in ASPEC are typed. However SPINE does not directly provide any type checking services for ASPEC. This is not a problem because all the type information in an ASPEC description is reflected in the C++ evaluator code (PINE) generated by SPINE. Hence, type-checking is done at the level of PINE code by a C++ compiler.

Currently, ASPEC only supports scalar data types: `integer, float, boolean, PooledString`[2] and C-like enumeration types. If one wants to support arbitrary data types one needs to be careful about two things: Firstly, synth-function memoization should be able to handle comparisons of arbitrary data types. We can easily achieve this by overloading the equality operator '==' in C++. Secondly, some form of garbage collection scheme (like reference counting) will have to be implemented. This is to ensure that memory allocated for a non-scalar attribute is recovered when the attribute is assigned a new value. For simplicity therefore, in the current implementation, SPINE does not provide support for complex data types in ASPEC.

5.8 Attribute Declarations

For each attribute of a phylum in the ASPEC the following information must be provided: What is its data type? Is it an inherited or synthesized attribute? Can its value be over-ridden by external sources?

SPINE currently does not use attribute over-ride information. But this feature will be eventually incorporated into SPINE, because there are applications like layout attribute computations where

[2] A PooledString class object differs from a "char*" or "const char*" in that it is guaranteed to be both pooled and permanent. By "pooled", we mean that if two PooledStrings have string value "foo", then they share storage for that string. By "permanent", we mean that the const char* returned by PooledString::chars access method will never change or be freed.

one might want to over-ride the value of a layout attribute to take on a forced new value.

Attribute declaration defaults can be specified at the start of the ASPEC description file. Whenever an attribute declaration has missing information, the defaults declarations will be used.

For example,

```
%% Default Attribute Declaration Section
  int inherited override
...

%% Attribute Declaration Section
phylum P, SL
with
  Width, Height:  synthesized:  nooverride;
  Top, Left;
  Bold:  bool;
end
```

In the above example, phylum `P` and `SL` are declared to have the same set of attributes. `Width` and `Height` are declared to be synthesized, non overridable attributes. All the other attributes are over-ridable and inherited by default. The `Bold` attribute is the only one of boolean type; all others are integer type by default.

Often all phyla in the grammar, with a few exceptions, have the same set of attribute declarations. There is a convenient short hand to capture this common case.

For example,

```
%% Attribute Declaration Section
phylum START
with
end

phylum STMT, STMT_LIST
with
  stmt_num:  int:  inherited;
  width, height:  float:  synthesized;
end

phylum *      /* matches any other phlya */
with
  width, height:  float:  synthesized;
  FontSize:  int, inherited;
  Bold, Italic:  bool:  inherited;
end
```

The * in the third (last) phylum attribute declaration will match with any phylum that does not already have an attribute declaration. In the above example the phylum `START` has no attributes, the phlya `STMT` and `STMT_LIST` have `stmt_num, width and height` attributes. All other phyla in the grammar have `width, height, FontSize, Bold and Italic` attributes.

5.9 Sanity Checks

Often users would like to incorporate sanity checks (like assert statements) in the AG specification on the values of attributes computed at a particular operator instance (node in a derivation tree). In ASPEC this can be done in the following manner: declare a dummy synthesized attribute, say `sanity`, for the left hand side phylum of the operator; now add a semantic rule for this attribute. The rule simply invokes an externally coded sanity check function and passes it the relevant parameters. In the current implementation any error reporting or corrective action has to be handled by the sanity function on its own.

For example, if for an operator say, `P --> A B` one wants to check if the `width` attribute of `P` is positive and is greater than the `width` attributes of both `A` and `B`:

```
P --> A B
{
  ...
  /* sanity:  a dummy synthesized attribute of P */
  $$.sanity =
      (isPositive($$.width) &&
      isGreater($$.width,$1.width) &&
      isGreater($$.width,$2.width));
}
```

5.10 Error Detection

Fairly useful and elaborate messages for errors in the ASPEC input are given by SPINE. The wide variety of errors detected by the system include: syntax errors, missing semantic rule for an output attribute of an operator, redeclaration of an attribute, multiple definitions for an output attribute of an operator, undefined phylum, illegal attribute reference in the right hand side of a semantic rule etc. In case the ASPEC input fails the SNC membership test, the operator (production) that has a circularity in its dependency graph, and a specific attribute that is involved in the circularity (by being transitively dependent on itself in the production's dependency graph) are reported.

6 Ensemble as an application for SPINE

SPINE has been put to practical use in the Ensemble research system being developed at Berkeley. Though SPINE has been built as a general purpose system for incremental evaluation of attribute grammars for any application, the motivation for the work actually stemmed from the need to improve the interactive performance of Ensemble's presentation system. This section details the various issues relating to the use of SPINE for Ensemble, describes the details of the PINE-Ensemble interface (see Section 6.5), and discusses the some of the limitations of using SPINE (Section 6.6). Section 6.7 gives some performance numbers.

6.1 Ensemble Overview

Ensemble is a software development environment that provides uniform language-based services for creation and manipulation of a wide variety of structured documents; programming language documents, structured natural language documents (e.g.: memos, articles, slides), structured graphics and structured video.

Ensemble takes the view that the content of a document and its presentation (the manner in which it is visually rendered to a user) are two clearly different aspects that need not be tightly coupled. This decoupling enables Ensemble to achieve elegantly its design goal of offering users the ability to manipulate multiple representations of a common underlying document. It is possible to support editing operations on each of the representations and have editing changes in one be reflected consistently in each of the others.

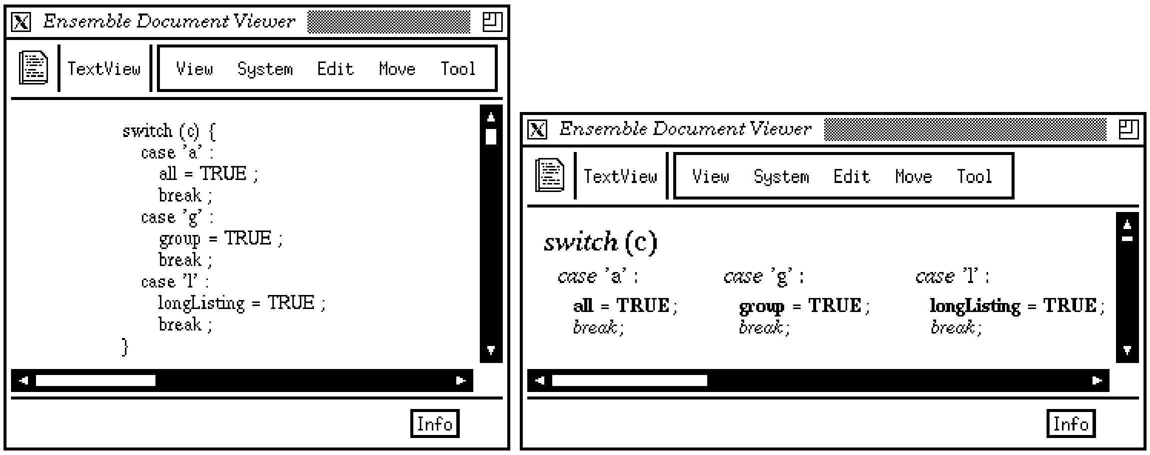

Figure 2: A two presentation example for switch statement

6.2 Presentations in Ensemble

The appearance of an Ensemble document is primarily defined by a formal specification called a *presentation schema*. Each Ensemble document has a particular structure i.e. it conforms to a context free grammar like the pascal language grammar or a grammar for articles etc. Users write one or more presentation schemas for each structured document type. A document corresponding to a particular structure can be viewed in as many styles as the number of presentation schemas written for that structure. Figure 2 illustrates this idea with two different presentations for the same `switch` statement document.

Each media is characterized by a set of *presentation attributes*. The process of formatting a document of a certain media type involves computation of the media specific presentation attributes for all sub-components of the document. For example, for a text media document, like the `switch` example, typical presentation attributes would be the font name, font size, position of a piece of text and so on. On the other hand a graphics media document would probably manipulate attributes like line width, fill color and fill pattern.

The presentation schema consists of a set of formatting rules that specify how the presentation attributes must be computed. When documents are edited the presentation attribute values at existing nodes of the document tree may become invalid. The tree itself can undergo changes during editing. Attribute reevaluation over the entire tree is a very costly operation even for reasonably small sized programs.

6.3 Performance bottlenecks in Proteus

Proteus is the subsystem of Ensemble that is responsible for computing values of presentation attributes. It adapts to the needs of a variety of different media. The first version of the Proteus exhibited very poor interactive performance. This could be attributed to two fundamental problems with the implementation:

- Attribute computations were not incremental. Every edit operation resulted in attribute computation of all attribute instances over the entire tree even though in most cases only a few attribute instances actually change value.
- The presentation schemas, written in a special language called the *presentation language*, were *interpreted* by Proteus. This involved building several run time data structures in the heap that represent the presentation rules. These data structures are interpreted by Proteus for every attribute computation. This is both space and time inefficient. This problem, though not as severe as the first one, is worth solving.

The goal for the new version of Proteus was clear: to overcome the aforementioned problems. We take advantage of the fact that all Ensemble documents have a tree representation. This enables us to write formal specifications of Ensemble documents using *attribute grammar* formalism. An AG specification in this framework would correspond to presentation attribute computations over the document tree structure. The advantage of using attribute grammars is that *incrementality* is implicit in the formalism.

6.4 The new approach

Under the new scheme, users write formal specifications of the presentation in the ASPEC attribute grammar description language. Each ASPEC input corresponds to a unique structure type, e.g., Fortran document type, article type etc. Multiple presentations for a particular structure type are simply described by means of multiple ASPEC specifications, all of which map to the same structure of documents, but contain different rules for presentation attribute computations.

These ASPEC specifications are transformed off-line by SPINE into PINE. The generated PINE code has embedded in it all the information needed to incrementally compute presentation attributes in accordance with the given input ASPEC description. This code is compiled to obtain an PINE object module that gets dynamically loaded in by the Ensemble system on demand.

The presentation information for Ensemble documents is represented as an attribution of the *presentation syntax tree*[3]. The PINE module is *stateless*: all the state information like the values of attribute instances, valid bits and values of old attributes for the purpose of caching resides entirely on the presentation tree. The PINE code computes values of attributes by accessing values of other attributes from the tree, stores new values for attributes in the tree, and updates semantic state information of the tree to obtain incrementality. It will soon be made clear why statelessness of PINE is a useful and desirable property.

Figure 3 presents a simplified view of the typical run-time interactions between the various components of Ensemble for the new version of Proteus that uses PINE modules for presentation

[3]We use the term presentation syntax tree instead of document syntax tree because these can often be different. For example, the presentation syntax tree could have more nodes corresponding to automatically generated text like page numbers. This is referred to as tree elaboration in the Ensemble context.

attribute computations. In the figure, `doc1.f77` and `doc2.f77` are both assumed to be documents that correspond the same structure, namely the Fortran grammar. We further assume that two ASPEC presentations specifications, `f77-p1.aspec` and `f77-p2.aspec` exist for Fortran, and PINE '.o' modules have already been generated for them off-line. If say the user wants to view each document in each of the given presentations, the PINE module corresponding to each of the presentation will get dynamically loaded in by Ensemble. The key observation here is that it suffices to load exactly one copy of each PINE module (e.g.: `f77-p1.o`) even if there are multiple number of documents (like `doc1.f77` and `doc2.f77`) that want to use that common presentation specification (`f77-p1.aspec`). This sharing of PINE modules between multiple documents is possible only because of their statelessness.

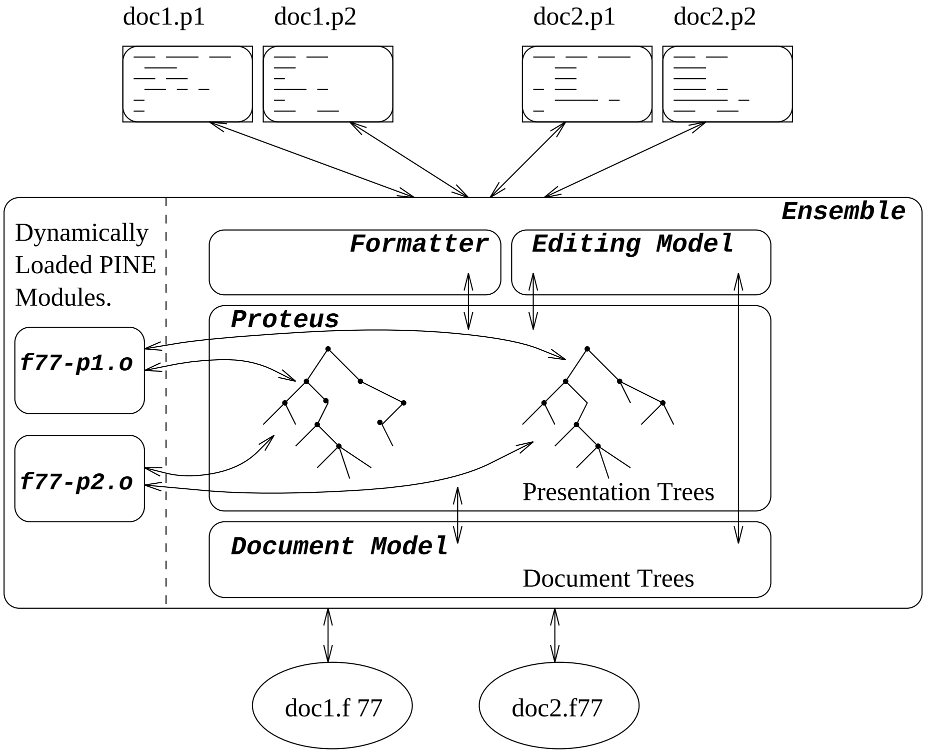

Figure 3: Simplified view of run-time interactions for a two document, two presentation example.

6.5 The PINE-Ensemble interface

Ensemble is quite a large software system, nearly 100,000 lines of code! It was decided that it would be simpler to adapt SPINE to match Ensemble's requirements rather than vice-versa. Hence, a specialized interface has been implemented for the purposes of using PINE code for Ensemble[4]. Changes in the interfaces were inevitable in any case, because of the need to support multiple presentations. This meant that on the same node different sets of attribute instances corresponding to different presentations could co-exist.

[4]The reader who is not interested in Ensemble specific implementation details may skip this section.

In the Ensemble world, tree nodes correspond to the `PNODE` class and the presentations to the `AgProteusPres` class. The `AgProteusPres` class, actually its parent class `ProteusPres`, implements all the navigation functions that required by the PINE (e.g.: get parent of node or get a child of a node) instead of these being implemented as virtual functions as in the case of our more general interface (see Section 4.3).

The "synth_functions" now take an additional argument, a pointer to the `AgProteusPres` class. This arguments tells PINE what the presentation for which the attribute is being computed is.

When a presentation is initialized it is necessary to dynamically load the appropriate PINE module if it has not already been loaded. This is done by invoking the `loadPine` method in the `AgProteusPres` class. Proteus needs to be aware of two function entry points in the loaded PINE module: `__update_RootPhylum` and `allocAttrData`. The first function handler corresponds to the function that triggers an attribute reevaluation in the tree. The arguments to the function are a pointer to the presentation and to the root node of the tree.

We will now discuss the need for the `allocAttrData` handler. The set of attribute instances that reside on a node depends on the type of the node (or phylum). All attribute instances on a node are collected up in a node specific C++ class object. SPINE generates definitions for these node specific classes in the PINE code, each of which inherits from a common base class. Objects of the derived class have storage for the values of attributes and other book keeping state information. Attribute values can be accessed through the base class by means of virtual functions. For every node, we need to be able to allocate an object the appropriate derived class. This is done using the `allocAttrData` function that is implemented in PINE. The node type must be passed as an argument. The name of the base class is medium dependent (_<*media_name*>`_PhylumData`). The preamble of the ASPEC file contains the required MEDIA declaration. The virtual functions in the class provide access to the medium specific presentation attributes.

When a document is edited the `subtreeChange` method in each of its presentations is invoked. A pointer to the edited subtree is passed as argument to this function in order to signal PINE of the change. There can be one or more `subtreeChange` calls before an attribute recomputation is initiated using a call to `reEval`. Both `subtreeChange` and `reEval` have been implemented in the `AgProteusPres` class.

6.6 Limitations of using AGs in formatting

There are some caveats in using a attribute grammar based presentation system for Ensemble. A user can overcome some of the limitations of the system by means of suitable tricks. In this way, the system can be used to the best possible extent.

- An inherent problem with attribute grammars is that all attributes dependencies are local. It may sometimes be the case that the value of an attribute at a node (say *dest*) actually depends on another attribute somewhere far away in the tree on another node (say *src*). In such a situation a special transfer attribute will have to be set up that simply propagates the value of one attribute by means of *copy rules* through a common ancestor of *dest* and *src* nodes.

- ASPEC specifications must conform to SNC class of attribute grammars. This does not appear to be a real restriction as the SNC class seems to incorporate almost all practical grammars. However an ASPEC input may fail the circularity test because of a certain unintentional typing error. SPINE helps the user rectify the error in such situations by pointing out the attribute involved in the circularity.

- Section 4.2.6 showed that the PINE based incremental evaluators can exhibit poor performance for unbalanced or degenerate trees. It is important to bear this in mind while writing grammar descriptions. Often structuring the grammar on an compact abstract syntax can considerably reduce the height of the tree. For example, repetitive constructs like a list of statements are traditionally represented as linearly recursive lists in order for the grammar to be non-ambiguous. This results in arbitrarily degenerate trees. A representation of sequence as a balanced binary tree [Mad93] in the abstract syntax has the effect of bounding the depth of the tree by $O(\log n)$. Although this introduces ambiguity in the grammar it provides the incremental parser more scope for effective reuse of tree sections.

- A more serious concern with the AG formalism is that only grammar preserving transformations of the tree structure are allowed. To be more precise, attribute instances on a tree can be assigned consistent and meaningful values only if the tree conforms to the grammar. Therefore, the attribute reevaluation process cannot be invoked after an editing operation that violates the grammar. Reevaluation can proceed only after a sequence of editing changes restore the tree to a legal configuration. This can be quite a problem during, for example, the document creation phase when for most of the time the document does not conform to its grammar. It is not clear that there is a simple solution to this problem. A possible method is to use heuristics to provide some default formatting. This might involve using the old value of an attribute if one exists, otherwise using some safe default value.

6.7 Performance Measurements

Some simple performance measurements have been done to determine the benefits of using the SPINE system in Ensemble. The results have been found to be extremely satisfactory. The tests were conducted on a Sparc-10 workstation. The document structure that we tested first was a C `switch` statement. The grammar corresponding to the switch statement had 23 non-terminal and terminal symbols and 22 productions.

Two ASPEC presentation descriptions were written for the switch grammar. Each one was about 150 lines long. Both presentations required about 17 presentation attributes to be computed at each node of the presentation tree.

We first measured the time required for off-line evaluator generation using SPINE. This includes time to parse the ASPEC input, do the SNC analysis and generate code. The real time for this off-line activity was found to be around 1.1 seconds for both presentations.

The interactive performance is obviously of greater interest. The sample switch statement document used in the test was about 20 lines of code comprising mainly of three *case* blocks each of which had three assignment statements followed by a *break* statement.

The metric used for evaluating the system is the maximum number of character presses that can be processed per minute. The disadvantage of this metric is that it is dependent on the location of the edit operation in the document. In fact, any form of performance measurement of an incremental attribute evaluator is tied to the specific edit changes that are being done since the

Table 1: Characters Processed per Minute: Small Document Example

# of presentations	*Old System*	*New System*	*Speedup*
1	88	264	3.0
2	66	230	3.5
3	31	146	4.7

Table 2: Characters Processed per Minute: Large Document Example

# of presentations	*Old System*	*New System*	*Speedup*
1	42	166	4.0
2	21	94	4.5
3	13	64	4.9

time for attribute reevaluation depends on the edit operation itself. However, the metric does gives us a feel for the potential speedups possible due to the use of SPINE in Ensemble.

Processing a character involves detection of a key press, an update to the document, computation of new presentation attributes and rerendering of the updated document. Though the performance enhancements due to use of the SPINE system only applies to the attribute computation stage, we would like to see its effect on the overall interactive system performance of Ensemble. Hence the motivation for choosing such a metric.

We compared the old system (non-incremental and interpreted) with the new one (incremental and compiled) for the switch document described before. We also varied the number of active presentation on the document. Table 1 summarizes the results of the studies. Similar measurements were also taken for a toy language document of nearly double the size of the `switch` document. The results are summarized in Table 2. All numbers are in characters processed per minute.

The old system is far more sensitive to changes in the document size because of its non-incremental nature. The speedups in the two tables above substantiate this claim. The larger the size of the document the greater is the speedup of the incremental system over the non-incremental one. The variation in speedups for a document when the number of presentations is increased also exhibits similar behavior.

7 Conclusions

SPINE is a system for generating incremental evaluators for attribute grammars. They are applicable to the broad class of SNC attribute grammars. SPINE system has a simple yet general interface that greatly simplifies the task of integrating SPINE into a wide variety of tree modifying client applications for the purpose of incremental attribute computations. Simplicity, close to optimal performance and ease of system integration make SPINE an ideal tool for incorporating incremental language-based services in software systems.

A reliable and robust version of the SPINE system exists currently. The system is about 7000 lines of C++, flex and bison code. SPINE has been integrated into Ensemble for incremental computation of presentation attributes of structured documents. The performance results have been very encouraging. We obtained speedups of 3 to 4 in overall system performance even on small documents in comparison to the previous non-incremental version. Work still needs to be done on providing support for overriding attribute values from outside of the attribute grammar system. This may involve propagation of new values to other attribute instances inside the attribute grammar.

Higher-order attribute grammars overcome some of the problems of traditional attribute grammars by allowing remote dependencies. This eliminates the need to have inefficient copy rules all over the tree. It would be interesting to see the performance improvements gained from using incremental attribute evaluation on these attribute grammars.

A limitation of the current implementation is that there is support for only scalar data types. This is not due to a limitation in the design of the incremental attribute evaluation technique. Once we incorporate a garbage collection scheme and support for equality comparisons for arbitrary data types it would be possible to use SPINE for applications like context-sensitive semantic analysis of languages. Also, there is scope for work on improving the language features of ASPEC; support for macros, assert statements, greater type safety and modularity.

In the SPINE model of attribute computations, whenever a request for attribute reevaluation is made, the entire tree is restored to a consistently attributed state. In many applications, at any given time, only a subset of the attributes are of interest. Evaluators that attempt to reevaluate only the set of demand attributes are called demand-driven or lazy evaluators. We are currently exploring ideas for lazy incremental attribute evaluation under SPINE.

8 Acknowledgements

I thank Michael A. Harrison, my research advisor, for his guidance, support and encouragement. I want to thank Susan Graham for her help, suggestions and support. I also wish to thank Kathy Yelick for her careful reading of the report. Special thanks to my fellow graduate students Ethan Munson, William Maddox and others in the Ensemble group for their help and ideas.

References

[CF82] B. Courcelle, and P. Franchi-Zannettacci, “Attribute Grammars and Recursive Program Schemes”, *Theoretical Computer Science*, Vol. 17, 1982, pp. 163-191, 235-257.

[Eng84] J. Engelfriet, “Attribute Grammars: Attribute Evaluation Methods”, *Methods and Tools in Compiler Construction*, Cambridge University Press, 1984.

[Fil87] G. File, “Classical and incremental attribute evaluation by means of recursive procedures”, *Theoretical Computer Science*, Vol. 53, 1987, pp. 25-65.

[Gra92] Susan L. Graham, “Language and document support in software development environments”, *Proceedings of the Darpa’92 Software Technology Conference, Los Angeles*, April 1992.

[Jou84] Martin Jourdan, "Strongly Non-Circular Attribute Grammars", *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, Vol. 19, No. 6, June 1984, pp. 81-93.

[Jou91] Martin Jourdan and Didier Parigot, "Internals and Externals of the FNC-2 Attribute Grammar System", *Attribute Grammars, Applications and Systems*, Lecture Notes in Computer Science 545, Springer-Verlag, June 1991, pp. 485-506.

[Kas80] U. Kastens, "Ordered Attributed Grammars", *Acta Informatica*, 13, 1980, pp. 229-256.

[Knu68] Donald E. Knuth, "Semantics of Context-free Languages", *Mathematics Systems Theory*, Vol. 2, (1968), pp. 127-145.

[KW76] K. Kennedy, and S. K. Warren, "Automatic Generation of Efficient Evaluators for Attribute Grammars", *3rd ACM POPL*, Atlanta, Georgia, Jan 1976, pp. 32-49.

[Mad93] William Maddox, and Susan L. Graham, "Efficient Incremental Attribute Evaluation", Submitted to PLDI'94, University of California, Berkeley, Nov 1993,

[Mun92] Susan L. Graham, Michael A. Harrison, and Ethan V. Munson, "The Proteus presentation system", *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, ACM Press, December 1992, pp. 130-138.

[Pec90] Stephen B. Peckham, "Incremental attribute evaluation and multiple subtree replacements", Technical Report TR 82-514, Ph. D. dissertation, Department of Computer Science, Cornell University, 1982.

[Rep83] Thomas W. Reps, Tim Teitelbaum, and Alan Demers, "Incremental Context-Dependent Analysis for Language-Based Editors", *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, July 1983, pp. 449-477.

[Rep89] Thomas W. Reps, and Tim Teitelbaum, *The Synthesizer Generator: A system for constructing language based editors*, Springer-Verlag, 1989.

[Vog91] Harald Vogt, Doaitse Swierstra, and Mattijs Kuiper, "Efficient incremental evaluation of higher-order attribute grammars", *Lecture Notes in Computer Science*, Vol. 528, Springer-Verlag, 1991, pp. 231-242.

A ASPEC Grammar

```
aspec_program :
        'MEDIA' mediaName
        '#include' '"'mediaPresIncludeFile'"'
        '%%'
        attrDeclDefaultsSection
        '%%'
        enumTypesSection
        '%%'
        attrDeclsSection
        '%%'
        prodnDefaultsSection
        '%%'
        phylumDefaultsSection
        '%%'
        startNode
        semanticRulesSection
        ;
mediaName : identifier
        ;
mediaPresIncludeFile : identifier
        ;
attrDeclDefaultsSection :
          attrDefaultQualifier attrDeclDefaultsSection
        | /* empty */
        ;
attrDefaultQualifier :
          'inherited'
        | 'synthesized'
        | 'override'
        | 'nooverride'
        | validType
        ;
enumTypesSection :
          enumType enumTypesSection
        | /* empty */
        ;
enumType : 'enum' enumTypeName '{' enumIdList '}' ';'
        ;
enumIdList :
          enumId ',' enumIdList
        | enumId
        ;
enumId : identifier
```

```
        ;
enumTypeName : identifier
        ;
attrDeclsSection :
          phylumAttrDecls attrDeclsSection
        | lastPhylumAttrDecls
        ;
phylumAttrDecls :
        'phylum' phylumList 'with' attrDeclList 'end'
        ;
lastPhylumAttrDecls
          'phylum' '*' 'with' attrDeclList 'end'
        | phylumAttrDecls
        ;
phylumList :
          phylum ',' phylumList
        | phylum
        ;
phylum : identifier
        ;
attrDeclList :
          attrDecl attrDeclList
        | /* empty */
        ;
attrDecl : attrIdList attrQualifierList ';'
        ;
attrIdList :
          attrId ',' attrIdList
        | attrId
        ;
attrId : identifier
        ;
attrQualifierList :
          ':' attrQualifier attrQualifierList
        | /* empty */
        ;
attrQualifier :
          attrDefaultQualifier
        | definedEnumId
        ;
prodnDefaultsSection :
          prodnDefaultRule prodnDefaultsSection
        | /* empty */
        ;
prodnDefaultRule :
```

```
          pDfltLAttrRef '=' cExpr ';'
        | iDfltLAttrRef '=' cExpr ';' optional_qualifier
        ;
pDfltLAttrRef :
          '$' '$' '.' attrId
        | '$' integer '.' attrId
        ;
iDfltLAttrRef : '$' 'i' '.' attrId
        ;
optional_qualifier :
          '[' range_index ',' range_index ']'
        | /* empty */
        ;
range_index :
          integer
        | '_n'
        ;
phylumDefaultsSection :
          phylumDefaults phylumDefaultsSection
        | /* empty */
        ;
phylumDefaults :
        phylumList ':' '{' phylumAttrRules '}'
        ;
phylumAttrRules :
          phylumAttrRule phylumAttrRules
        | /* empty */
        ;
phylumAttrRule :
        attrId '=' cExpr ';'
        ;
startNode : '%start' rootPhylum
        ;
rootPhylum : identifier
        ;
semanticRulesSection :
          production semanticRulesSection
        | /* empty */
        ;
production :
        phylumName '-->' phylumNames '{' prodnAttrRules '}'
        ;
phylumNames :
          phylumName phylumNames
        | phylumName
```

```
        | /* empty */
        ;
phylumName : phylum
        ;
prodnAttrRules :
          prodnAttrRule prodnAttrRules
        | /* empty */
        ;
prodnAttrRule : lAttrRef '=' cExpr ';'
        ;
lAttrRef :
          '$' '$' '.' attrId
        | '$' integer '.' attrId
        ;
cExpr : term cExpr1
        ;
cExpr1 :
          '+' term cExpr1
        | '-' term cExpr1
        | '/' term cExpr1
        | '*' term cExpr1
        | /* empty */
        ;
term :
          '(' cExpr ')'
        | '(' cExpr relop cExpr ')'
        | '-' term
        | '!' term
        | function '(' cExprList ')'
        | function '(' ')'
        | 'if' cExpr 'then' cExpr 'else' cExpr 'fi'
        | integer
        | float
        | string
        | bool
        | definedEnumId
        | attrRef
        ;
attrRef :
          '$' '$' '.' attrId
        | '$' integer '.' attrId
        | relAttrRef
        | iAttrRef
        | '@' function '.' attrId
        ;
```

```
relAttrRef :
          '$' / integer '.' attrId
        | '$' / '+' integer '.' attrId
        | '$' / '-' integer '.' attrId
        ;
iAttrRef :
          '$' '(' iExpr ')' '.' attrId
        | '$' 'i' '.' attrId
        ;
iExpr : iTerm iExpr1
        ;
iExpr1 :
          '+' iTerm iExpr1
        | '-' iTerm iExpr1
        | /* empty */
        ;
iTerm :
          'i'
        | integer
        | '(' iExpr ')'
        | '-' iTerm
        ;
cExprList :
          cExpr ',' cExprList
        | cExpr
        ;
bool :
          'true'
        | 'false'
        ;
relop :
          '=='
        | '>'
        | '<'
        | '>='
        | '<='
        | '&&'
        | '||'
        ;
validType : identifier
        ;
definedEnumId : identifier
        ;
function : identifier
        ;
```

B An example ASPEC input

```
MEDIA Text
#include "TextPresentation.h"

%% /* Attribute Declaration Defaults */
     inherited int override

%% /* Enum Types Section */
     enum visible_type {YES, NO};

%% /* Attribute Decls Section */

phylum START
with
end

phylum P
with
        Left, Top;
        StmtNum : synthesized;
        Width, Height : synthesized;
end

phylum S, D, A
with
         Left, Top;
         StmtNum : inherited;
         Width, Height : synthesized;
end

phylum *       /* for all other phylum use this declaration */
with
        Left, Top;
        Width, Height: synthesized;
        Bold : bool;
        Visible : visible_type;
end

%% /* Production Defaults Section */
        /* $$ --> $1 $2 ... $_n */

        $$.Width  = @AllChildren.Width;
        $$.Height = @AllChildren.Height;
        $i.Visible = NO;                        [1,_n]
```

```
        $i.Bold = false;                            [1,_n]
        $i.Left = $$.Left;                          [1,_n]
        $1.Top  = $$.Top;
        $i.Top  = $(i-1).Top + $(i-1).Height;   [2,_n]

%% /* Phylum Defaults Section */

VAR, SEMI, EQ, EXPR :
{
        Left = $/-1.Left + $/-1.Width;
}
A, D :
{
        StmtNum = $$.StmtNum;
}

%% /* Semantic Rules Section */

%start START   /* name of the root phylum */

START --> P  {
       $1.Left = 0;
       $1.Top  = 0;
}
P --> P S {
       $$.StmtNum = $1.StmtNum+1;
       $2.StmtNum = $1.StmtNum+1;
}
P --> S {
       $$.StmtNum = 1;
       $1.StmtNum = 1;
}
S --> A { }
S --> D { }

A --> VAR EQ EXPR SEMI {
       $1.Bold = if (5+4*3 > 4) then
                       true
                 else
                       false
                 fi;
}
D --> INT VAR SEMI { }
D --> STR VAR SEMI { }
```