SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 1(1), 1–4 (JANUARY 1988)

Modeling User-Provided Whitespace and Comments in an Incremental SDE

TIM A. WAGNER*
Reasoning, Inc.

SUSAN L. GRAHAM
University of California, Berkeley

SUMMARY

The handling of whitespace in batch environments is well understood for a variety of programming languages and whitespace models. However, *incremental* environments have not provided a satisfactory solution to this problem, due to the fact that this material must be included (and incrementally maintained) in the persistent representation, rather than simply discarded. We describe a combination of representation and analysis techniques that together provide generic support for explicit whitespace and similar 'extra-grammatical' material in an incremental, interactive software development environment. Our methods are independent of the lexical model, handling whitespace-insensitive languages (such as Fortran77), whitespace-sensitive languages (such as C++), and mixed-sensitivity languages (such as Haskell). The representation is uniformly structural: whitespace is integrated with the persistent program structure without introducing special cases. The representation is also efficient, imposing minimal overhead and guaranteeing logarithmic access times to *all* nodes. Two simple strategies for incrementally maintaining the proposed structure are described, one based on grammar transformation and the other on modifying the incremental parser.

KEY WORDS Whitespace Software development environments Incremental analysis Incremental parsing Structural representation Comments Language description

INTRODUCTION

Whitespace plays several roles in programming languages: it separates other tokens, provides the programmer with a degree of control over the visual presentation of the source code, and in some languages even serves as a syntactic construct. Batch compilers and environments handle whitespace in a simple manner, discarding it as the lexical analyzer scans each region of the text.[†] *Incremental* environments,

* This research has been sponsored in part by the Advanced Research Projects Agency (ARPA) under Grant MDA972-92-J-1028, and in part by NSF institutional infrastructure grant CDA-8722788.
The content of this paper does not necessarily reflect the position or policy of the U. S. Government.
Authors' addresses: Tim A. Wagner, Reasoning, Inc., 700 E. El Camino, Suite #300, Mountain View, CA 94040 and Susan L. Graham, 771 Soda Hall; Department of Electrical Engineering and Computer Science, Computer Science Division, University of California, Berkeley, CA 94720-1776.
email: `twagner@reasoning.com`, `graham@cs.berkeley.edu`
URL: `http://http.cs.berkeley.edu/~twagner`, `http://http.cs.berkeley.edu/~graham`

† Languages in which whitespace can play a syntactic role naturally require additional communication between the lexer and parser in some cases.

CCC 0038–0644/88/010001–04

Received 1 March 1988
Revised 25 March 1988

however, have typically failed to provide an adequate solution for handling whitespace.[1-3] When these environments are also multilingual, the need to simultaneously support multiple whitespace models exacerbates the problem. The lack of incremental, language-independent support for whitespace has limited both the scope and functionality of these environments.

One reason for this limitation is that the program representation in an incremental environment is both *persistent* and *structural*, requiring a different handling of whitespace than the simple approach taken in batch environments. Also, many interactive environments support high-quality program presentation services, where layout can be derived rather than forcing the programmer to express it solely through whitespace characters embedded in the program content. However, even in this setting, the programmer must be permitted to override the standard layout in order to express semantically significant layout decisions in a persistent fashion, requiring implicit and explicit whitespace to coexist.

In addition to the use of whitespace to influence the visual appearance of the program, the program representation must typically support many other elements that stand in a similar relation to the normal program structure: text-based comments,* constructs from embedded languages, transient representations of edited text (such as inserted text not yet incorporated by incremental analysis), and 'errors' in the form of material that was not successfully incorporated into the program structure during some previous incremental analysis. Supporting *any* of these elements requires the environment to address the same representation and reconstruction issues as with explicit whitespace.

In this paper we describe a simple integration of whitespace material with the persistent structural representation of the program. Our representation is independent of the whitespace model, and therefore of the language itself. The uniformity of the integration provides significant leverage: existing tools can easily view or ignore whitespace without introducing special cases, both structural and textual views of the program are supported, and all editing, change reporting, versioning, and analysis/transformation services treat whitespace material in the same manner as other program components. Our representation is also efficient, imposing minimal space overhead and guaranteeing logarithmic access times to all whitespace tokens by representing each contiguous sequence as a balanced binary tree.

By specifying only the structure of the program's representation, this method is independent of the lexical specification and analysis mechanisms that describe and create whitespace tokens. It is also compatible with a wide variety of presentation services and approaches: layout can be based solely on whitespace material embedded in the source text, completely independent of it, or use some combination of the two methods.[4]

Unlike batch systems, which rely primarily on the *lexer* to handle whitespace (where it is discarded once recognized), incremental systems use the incremental *parser* to integrate this material into the persistent program structure. We describe two methods for constructing our representation. Both support declarative language definition, an unrestricted editing model, and highly efficient incremental reconstruction of the program representation to incorporate the user's modifications.

The first method is based on grammar transformation. Each language supported by the environment is specified by a high-level ('declarative') description, which includes a syntactic specification in the form of a grammar. These grammars are typically written in an extended formalism and transformed internally to a simplified canonical form before being used to generate parse tables. Along with existing transformations, such as the flattening of regular-right-part notation, assumptions regarding the appearance of whitespace can be made explicit: the transformed grammar will include a (possibly empty) sequence of whitespace tokens between terminals in the source grammar. Common classes of grammars used for deterministic parsing (LALR(1), LL(1)) are closed under this transformation. By

* The need for environments to support existing programs and to interact with external tools will require them to support text-based comments in addition to structural annotations for the foreseeable future.

permitting whitespace to be mentioned in the *original* grammar, this approach can also be used with languages like Haskell, which allow whitespace to play a syntactic role in certain contexts.

The second approach encodes the additional instructions needed to build the whitespace representation directly into the incremental parsing algorithm. Existing incremental parsing algorithms based on either state-matching or sentential-form parsing can be extended easily to construct this representation without compromising incremental performance.

PROGRAM REPRESENTATION

We first review a program representation suitable for an incremental software development environment (ISDE), then describe enhancements that allow whitespace and similar program elements to be present in the representation. Hereafter, the term 'whitespace' is used to describe *any* material logically present in the token stream but not regarded as terminal symbols in the grammar. (The treatment of mixed-model languages, such as Haskell, is deferred until the following section.)

A Persistent, Language-Independent Program Representation

The *Ensemble* ISDE[5, 6] models all documents as persistent, primarily tree-like structures. The structure of the document corresponds to the abstract syntax tree of the program; the terminal symbols are tokens that collectively represent the program text.

Ensemble is designed to be multi-lingual. This is accomplished by describing each language using declarative specifications, which are compiled into efficient, specialized analyzers that can be dynamically loaded to customize a running environment. Existing language support in our environment includes C, Modula-2, Java, and Fortran. Language-specific tools provide incremental services, including lexing, parsing, semantic analysis, correctness-preserving transformations, and specification-driven presentation.

The syntax of each language is described to the environment by means of a grammar. In the program representation, each node represents an instance of a production in this grammar. This permits nodes to be strongly-typed (resulting in a space-efficient representation) and to provide run-time type information to client tools and services, which are themselves derived from formal specifications based on the grammar of the language.

The entire program representation, including its content, structure, and any annotations is *persistent*.[7] Versioning services provide efficient access to multiple versions simultaneously and enable modifications to be incrementally reported to all tools within the environment.

Integrating Whitespace with the Program Structure

Our whitespace representation was designed to fulfill a number of requirements:

Ordering: Whitespace must appear correctly ordered with respect to other program elements.

Uniformity: Clients and services such as editing, change reporting, and incremental analysis should have a single method for viewing the structure and content of programs.

Efficiency: There should be no limit on the number of contiguous whitespace tokens, and clients should be able to access each token in time logarithmic in the length of the sequence.

Abstraction: Clients should be able to view the program (or any subset of it) with or without whitespace, and both views should be equally efficient to produce.

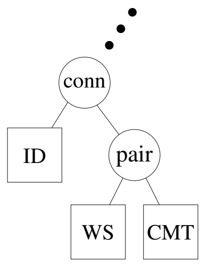

Figure 1. Our structural representation of whitespace material. A contiguous sequence of extra-grammatical tokens, which can include whitespace, text-based comments, unincorporated text, errors, and similar elements, are grouped and connected to the preceding token that represents a terminal symbol in the grammar.

Flexibility: The representation should be independent of the language, the whitespace model, the number and types of whitespace categories, and the mechanisms by which they are specified and discovered during lexical analysis.

The implementation is straightforward: each contiguous sequence of whitespace tokens is represented as a balanced binary tree. (Often this 'tree' will consist of a single whitespace token.) The sequence is logically associated with the preceding token that represents an instance of a terminal symbol in the grammar. An additional *connector node* is introduced to serve as the parent of the both the preceding terminal and the root of the whitespace sequence. Figure 1 illustrates the arrangement.

This approach meets all the criteria described above. Ordering is obviously maintained; the original program text can be reconstituted simply by walking the tree, inspecting the content of each lexeme encountered. Clients can navigate whitespace sequences with the same structural operations used elsewhere in the program, and whitespace tokens are implemented in the same fashion as other nodes.

Most services that view or update the program representation rely on the ability to navigate a select portion of the program structure efficiently. These services include editing, change reporting, incremental parsing, presentation, etc. While the average number of whitespace tokens following a given terminal symbol is low, individual sequences can become quite lengthy, as Figure 2 illustrates. The use of a balanced representation to implement a contiguous whitespace sequence allows any token in that

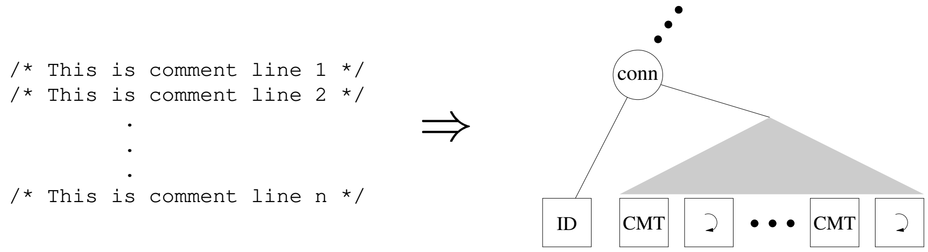

Figure 2. The representation of contiguous whitespace tokens. To prevent long sequences from degrading performance, a balanced binary tree is used to guarantee logarithmic access time to each token. (Curly arrows represent newlines.)

```
assign  →  id = exp ;
  expr  →  expr && expr
        |  expr || expr
        |  id
        |  . . .
  . . .
```

```
x␣=␣a&&b||
        c&&d;
```

```
x=a&&b||c&&d;
```

Figure 3. Running example. From left to right: grammar, program text including whitespace, program text without whitespace. (Curly arrows represent newlines, horizontal arrows denote tabs.) Figures 4 and 5 illustrate structural views of this example.

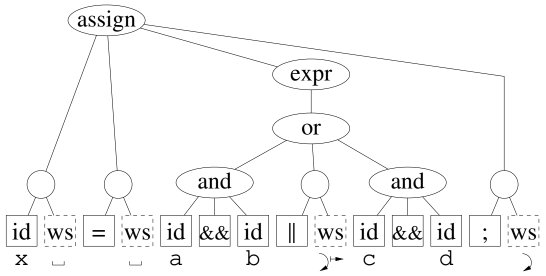

Figure 4. The structural representation of the program text (concrete syntax) *in Figure 3.*

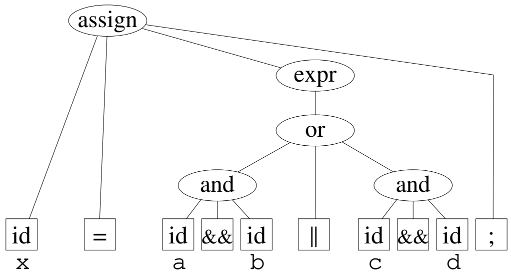

Figure 5. View of the concrete syntax with whitespace removed. Additional elements of the concrete syntax, such as punctuation, could be removed by further abstracting the structure.

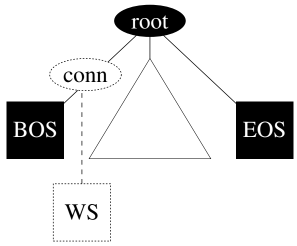

Figure 6. Sentinels used in program representation. The beginning of the token stream is marked by the `bos` *token, the end by* `eos`*. A new top-level root node is added to point to these two tokens and to the original root of the program representation. Any whitespace preceding the first terminal symbol in the program is connected to* `bos`.

sequence to be accessed in logarithmic time.* When other sequences in the program structure, such as declaration and statement lists, are similarly balanced, any node in the program can be accessed in time logarithmic in the length of the program.[8]

As with other sequences of program elements represented as balanced binary trees, the balancing condition is restored at each *commit point*, when the program is in a consistent state. (Each user edit and each incremental analysis are terminated by a commit operation.) The balancing condition is *not* imposed during editing or incremental parsing; newly-inserted tokens are held in a left- or right-recursive chain and are only balanced at the conclusion of the analysis.[9]

In our environment, most tools view the structural representation through an *abstraction* that tailors the tree view for their particular needs. Analysis tools, including the incremental lexer and parser, use abstraction in order to access versions of the program other than the current one. Other tools, such as presentation, use abstraction to customize their view of the program content. Abstractions are implemented in a stateless 'navigational' style.

The visibility of whitespace material is also handled by the abstraction mechanism. Since all nodes—including whitespace nodes—are typed, an abstraction can easily filter out unwanted material based on the node type. In the particular case of whitespace, tools that want to eliminate whitespace can do so by using an abstraction that treats each connector node as a proxy for its first child. Figures 3 through 5 illustrate both textual and structural views of a sample program under different abstractions.

Since we connect each whitespace sequence to the previous terminal symbol, whitespace material appearing at the beginning of the program constitutes a special case. To handle this, and to simplify the persistent representation of the program in general, we add three *sentinel nodes*. These serve as the initial and final tokens and the top (root) node in the program structure; Figure 6 illustrates the arrangement. Whitespace at the beginning of the program follows the beginning-of-stream sentinel token (`bos`), maintaining uniformity in the representation.

The presence of whitespace does not affect the editing model. Text inserted between existing tokens can be represented as an extension to one of the tokens or as a new 'unincorporated insertion' token placed between them and represented in the same fashion as whitespace tokens.†

* Since the token is the unit of granularity for both re-analysis and representation, the language description writer must take care to ensure that lexemes are short in practice. For instance, if whitespace or textual comments in a particular language typically span many lines, each line may be described as a separate token to increase the effectiveness of balancing.

† One complication in a completely whitespace-insensitive language is that whitespace elements can occur *within* an otherwise-atomic lexeme. This is most easily handled by providing two distinct views for each token's lexeme: a *verbatim* view appropriate for re-creating the exact textual content of the program and a *filtered* view that represents the lexeme as seen by most clients, with any whitespace characters removed.

root	$\rightarrow$	bos assign eos	*Add sentinel root.*
assign	$\rightarrow$	id = expr ;	*(Original start symbol)*
expr	$\rightarrow$	expr && expr	
	|	expr || expr	
	|	id	
...			
ws-pair	$\rightarrow$	WS WS	*Pair productions are shared*
	|	WS ws-pair	*by all whitespace sequences.*
	|	ws-pair WS	
	|	ws-pair ws-pair	
id	$\rightarrow$	ID	*For each terminal symbol,*
	|	id-conn	*allow a whitespace connector*
id-conn	$\rightarrow$	ID WS	*in its place.*
	|	ID ws-pair	
and	$\rightarrow$	AND	
...			

Figure 7. Whitespace support through grammar transformation. The grammar of Figure 3 is transformed to allow each terminal symbol to be followed by an optional sequence of whitespace tokens. `WS` *and* `ID` *represent whitespace and identifier tokens, respectively. The transformation applied to identifiers would be repeated for each terminal symbol* (`equals`, `and`, etc.) *and* `bos`.

When multiple categories of whitespace tokens exist, clients may require an efficient access path to the subset of tokens from a particular category within a heterogeneous whitespace sequence. To make this filtering process efficient, we can annotate each node in the binary tree representing the whitespace sequence with information that summarizes the union of the categories of the tokens in its yield. (Equivalently, the namespace of node types can be extended to convey this information, with the appropriate type selected whenever a structural editing operation occurs.) As with other summary information about the descendants of a node, category information can be incrementally maintained as the program structure is modified during editing, incremental parsing, or other transformations.

CONSTRUCTION METHOD I: GRAMMAR TRANSFORMATION

The representation described in the previous section can be constructed in various ways. In this section we describe a method based on *grammar transformation*, where the presence of whitespace is (usually) implicit in the source grammar supplied by the language description writer, but is made explicit in the transformed grammar used to generate the incremental analysis tools. In the following section, an alternate method is presented that instead modifies the incremental parsing algorithm and leaves the grammar unchanged.

The meta-language used to write grammars for language descriptions provided to the ISDE is typically an extended BNF; often regular expressions are permitted on the right-hand side of productions. In order to generate incremental analysis and transformation tools, the environment will translate the extended grammar to a canonical form. As part of this or a subsequent transformation, we can convert the *implicit* presence of whitespace to an *explicit* representation in the transformed grammar.* Figure 7 provides a template for this expansion, illustrating how one terminal symbol from our running example is re-written to permit a sequence of whitespace tokens to follow it.

* This transformation is similar to that proposed by Visser,[10] although no attempt is made there to produce an incremental evaluator from the result. The idea is also similar to the 'expected' comments in the Eiffel grammar.[11]

Each terminal symbol must be transformed in a similar fashion, introducing an accompanying connector production. In addition, `bos` must be similarly transformed and a production added to represent the root sentinel. (Since whitespace is attached to the preceding terminal symbol, `eos` does *not* require this transformation.) The four productions used to define `ws-pair` are included only once. These productions are written in a form that expresses the inherent associativity in the sequence, allowing a balancing algorithm to be employed. This notation will generate conflicts when the grammar is compiled by a deterministic parser generator such as `yacc` or `bison`. The default conflict resolution schemes ('prefer-shift', 'prefer-earlier-rule') can be safely used to produce a correct parser. (The same approach should be used when constructing parse tables for a GLR or other non-deterministic parser, to prevent the parser from constructing alternative interpretations of each whitespace sequence.)

Classes of grammars typically used for deterministic parsing (LR(k), LALR(k), and LL(k) for $k \geq 1$) are closed under this transformation, since the set of whitespace tokens is mutually exclusive with the grammar's original set of terminal symbols, and an entire whitespace sequence can be constructed using a single item of lookahead.

The grammar transformation approach is preferable when the compilation of language descriptions is easily extended. The environment *per se*, including the incremental parser, is unchanged. This approach is also preferable when it is necessary to describe languages in which whitespace can play a syntactic role, such as Haskell's 'off-side' rule.[12] In such languages, whitespace tokens will appear explicitly in the grammar; names for explicitly-mentioned whitespace tokens must be integrated with names introduced by the transformation process.*

CONSTRUCTION METHOD II: PARSER MODIFICATION

The grammar transformation described in the previous section is a simple mechanism for building the whitespace representation. In some circumstances, however, existing parse tables may need to be retained, or it may be appropriate to avoid additional transformations. Thus we present a second method for building this representation, based on directly modifying an incremental shift/reduce parser.

Rather than modify the grammar (and therefore the parse tables), we can simulate the additional shift and reduce actions by directly modifying the parsing algorithm instead—the `next_action` method is extended to handle the necessary operations as special cases. If no special rules apply, then this method will simply interrogate the parse table for the next action to take. The top two nodes on the parse stack and the lookahead symbol are sufficient to determine the next action in all cases; we do not change the parse state when shifting whitespace-related nodes. Note that this method, like the previous one, will shift a subtree that represents a whitespace sequence in constant time when it appears in the parser's input stream. Figure 8 summarizes the necessary changes to the incremental parser's `next_action` routine.

The parser-based approach can be implemented directly in both sentential-form and state-matching incremental parsers. A similar technique can be used to extend an incremental GLR parser by encoding a synthetic 'whitespace state' in the nodes of its graph-structured stack.[13]

When parsing is complete (the parse table indicates an *accept* action), the parse stack will contain two elements. The first is either the `bos` token itself or a connector node whose left child is `bos`. The second parse stack entry will be a node corresponding to the start symbol of the grammar. At this point

* Even with the ability to name whitespace explicitly in the grammar, Haskell is non-trivial to describe in a manner suitable for incremental analysis. The lexical description must discover indentation changes and encode them in the set of whitespace tokens, and the language specification must accommodate the possibility of both explicit and implicit (whitespace-based) scope delimiters. The latter violates the assumption that the original set of terminal symbols is mutually exclusive with the set of whitespace tokens. Incremental GLR parsing[13] can be used to overcome any conflicts induced by relaxing this restriction, using non-deterministic parsing to try several possibilities simultaneously.

```
ACTION IncrementalParser::next_action (NODE *lookahead, int parseState) {
  Lookahead is either bos or a CN node that wraps bos. Shift into the parser's initial state.
  if (stack.is_empty()) return SHIFT 0;

  The top two nodes on the stack and the lookahead symbol are sufficient to 'parse' whitespace.
  NODE *tos = stack.element(0), *prev;
  int tosType = tos→type, prevType = BadType, laType = lookahead→type;
  if (stack.depth() > 1) {prev = stack.element(1); prevType = prev→type;}

  Previous/current item are one of <WS WS>, <WS PR>, <PR WS>, <PR PR>. Reduce them to a PR node.
  if ((tosType == WS || tosType == PR) && (prevType == WS || prevType == PR))
    return REDUCE PR_RULE;

  Create a wrapper when the whitespace sequence is complete.
  if (tosType == PR && laType != PR && laType != CN && laType != WS)
    return REDUCE CN_RULE;

  Treat an unmodified connector appearing as lookahead as its first child.
  if (laType == CN)
    return next_action(lookahead→first_child(), parseState);

  Backtrack from reductions to expose a terminal symbol when necessary to connect new adjacent whitespace tokens.
  if (is_nonterminal(tos) && tosType != PR && (laType == PR || laType == WS))
    right_breakdown();

  Now it's safe to shift PR and whitespace tokens; note that the parse state remains unchanged.
  if (laType == PR || laType == WS) return SHIFT parseState;

  The 'grammatical' case: look in the parse table to decide on the action.
  return parseTable→next_action(parseState, lookahead);
}
```

Figure 8. Extensions to the incremental parser's `next_action` *method. Existing whitespace subtrees are shifted onto the parse stack without changing the parse state. New whitespace material is 'parsed' to create a left-recursive chain that will be (re)balanced when parsing is complete. When a connector node appears as the parser's lookahead symbol, its first child is used to determine the next parse action. Connecting whitespace to the preceding terminal symbol may require that terminal symbol to be exposed by removing the right-hand edge of the enclosing subtree; this is accomplished by the* `right_breakdown` *routine, shown in Figure 9. Its effect is illustrated in Figure 10. '*`WS`*' is used as a shorthand to represent all whitespace token types, which are treated as an equivalence class by this algorithm. 'CN' and 'PR' denote connector and pair nodes, respectively (Figure 1).*

```
Remove any subtrees on top of parse stack with null yield, then break down right edge of topmost subtree.
void IncrementalParser::right_breakdown () {
  NODE *node;
  do {  Replace node with its children.
    node = parseStack→pop();
    Does nothing when child is a terminal symbol.
    foreach child of node do shift(child);
  } while (is_nonterminal(node));
  shift(node);  Leave final terminal symbol on top of stack.
}

Shift a node onto the parse stack and update the current parse state.
void IncrementalParser::shift (NODE *node) {
  parseStack→push(parseState, node);
  parseState = parseTable→state_after_shift(parseState, node→symbol);
}
```

Figure 9. Procedure to break down the right-hand edge of the subtree on top of the parse stack. On each iteration, `node` *holds the current top-of-stack symbol. Any subtree with null yield appearing in the top-of-stack position is removed in its entirety.*

Initial configuration:

$\ldots A$ | `WS ...`

First iteration:

$\ldots BCE$ | `WS ...`

Fourth iteration:

$\ldots BCHI$ | `WS ...`

After breakdown is complete:

$\ldots BD$ | `WS ...`

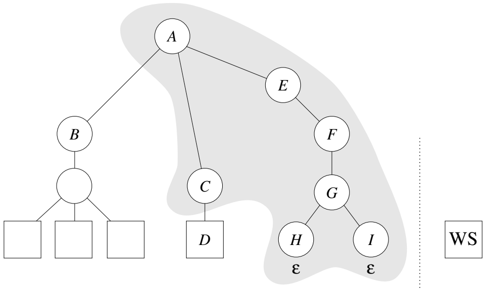

Figure 10. Illustration of right-breakdown. The shaded region shows the reductions 'undone' by the breakdown—all nodes representing reductions predicated on the following terminal symbol are removed. Any subtrees with null yield are discarded, then the right-hand edge of the subtree on top of the stack is removed, leaving its final terminal symbol in the topmost stack position. (The parse stack holds both states and nodes; only node labels are shown here.) After `right_breakdown` *has exposed the previous terminal symbol, the whitespace token in the lookahead position can be shifted onto the parse stack. The reductions represented by the removed nodes are simply regenerated by the incremental parser once the whitespace sequence has been integrated.*

the lookahead symbol, `eos`, can be pushed onto the stack and the three stack elements 'reduced' to create the sentinel root node of the program representation (Figure 6).

Error *detection* is essentially unchanged by the presence of whitespace; any whitespace tokens immediately preceding the point of detection will typically be gathered into their sequence representation before the error is discovered. Incremental error *recovery*[14] is beyond the scope of this paper, but the presence of whitespace structure does not require significant changes to either correcting or non-correcting strategies.

The representation constructed by modifying the incremental parser is structurally equivalent to that produced by grammar transformation up to the balancing of sequences; the only difference is that a *single* connector type can be used in this approach, since there is no need to ensure closure properties required by the transformation-based scheme.

CONCLUSION

Supporting explicit whitespace and other program elements that do not appear in the grammar of the language is a fundamental requirement for any tool or environment that must model software artifacts. We have presented a simple representation suitable for use in environments where the program representation is primarily structural and is both persistent and *incrementally* maintained. Two methods for constructing this representation are supplied: one based on a grammar transformation that can be used with all common grammar classes, and a second based on simple extensions to existing incremental parsing algorithms. The result is a flexible, uniform, and scalable method for handling extra-grammatical program elements in a language-independent fashion.

REFERENCES

1. Rolf Bahlke and Gregor Snelting, 'The PSG system: From formal language definitions to interactive programming environments', *ACM Trans. Program. Lang. Syst.*, **8**(4), 547–576 (1986).
2. Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter, 'The Pan language-based editing system', *ACM Trans. Softw. Eng. and Meth.*, **1**(1), 95–127 (1992).
3. Thomas W. Reps and Tim Teitelbaum, *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag, Berlin, 1989.
4. Mark van den Brand and Eelco Visser, 'Generation of formatters for context-free languages', *ACM Trans. Softw. Eng. and Meth.*, **5**(1), 1–41 (1996).
5. Tim A. Wagner, 'Practical algorithms for incremental software development environments', *Ph.D. Dissertation*, University of California, Berkeley, 1997. Available as technical report UCB/CSD–97–946.
6. Vance Maverick, 'Presentation by tree transformation', *Ph.D. Dissertation*, University of California, Berkeley, 1997. Available as technical report UCB/CSD–97–947.
7. Tim A. Wagner and Susan L. Graham, 'Efficient self-versioning documents', *CompCon '97*, San Jose, Calif., Feb. 1997, pp. 62–67. IEEE Computer Society Press.
8. Neal M. Gafter, 'Parallel incremental compilation', *Ph.D. Dissertation*, University of Rochester, Rochester, N.Y., 1990.
9. Tim A. Wagner and Susan L. Graham. Efficient and flexible incremental parsing, 1996. Submitted to *ACM Trans. Program. Lang. Syst.*
10. Eelco Visser, 'A family of syntax definition formalisms', *Proceedings of ASF+SDF95—A Workshop on Generating Tools from Algebraic Specifications*, May 1995.
11. B. Meyer, *Eiffel: The Language*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1992.
12. Paul Hudak et al., 'Haskell report', *SIGPLAN Not.*, **27**(5), R (1992).
13. Tim A. Wagner and Susan L. Graham, 'Incremental analysis of real programming languages', *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*. ACM Press, Jun. 1997, pp. 31–43.
14. Tim A. Wagner and Susan L. Graham. History-sensitive error recovery, 1997. In preparation.