Grammatical Abstraction and Incremental Syntax Analysis in a Language-Based Editor*

Robert A. Ballance†, Jacob Butcher, and Susan L. Graham

Computer Science Division—EECS
University of California, Berkeley
Berkeley, California 94720

Abstract

Processors for programming languages and other formal languages typically use a concrete syntax to describe the user's view of a program and an abstract syntax to represent language structures internally. Grammatical abstraction is defined as a relationship between two context-free grammars. It formalizes the notion of one syntax being "more abstract" than another. Two variants of abstraction are presented. Weak grammatical abstraction supports (i) the construction during LR parsing of an internal representation that is closely related to the abstract syntax and (ii) incremental LR parsing using that internal representation as its base. Strong grammatical abstraction tightens the correspondence so that top-down construction of incrementally-parsable internal representations is possible. These results arise from an investigation into language-based editing systems, but apply to any program that transforms a linguistic object from a representation in its concrete syntax to a representation in its abstract syntax or vice versa.

*Sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871 (monitored by Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089), by IBM under IBM Research Contract No. 564516, and by the State of California MICRO program.. Robert A. Ballance was supported in part by a MICRO fellowship.

†Current address: Department of Computer Science, University of New Mexico, Albuquerque, New Mexico 87131.

Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation
Atlanta, Georgia, June 22–24, 1988

1 Introduction

Language-processing tools that parse text often require two different descriptions of a language. The abstract syntax defines the structures of the language having independent semantic relevance. The concrete syntax defines the structures of the language as seen by a user and as required by a parser. Either can be used to describe an internal (tree-structured) representation of a linguistic object within a language-processing system.

In a language-based editing system, both text- and structure-oriented operations play a role. Our premise is that while using a language-based editor, the user must be able to regard linguistic objects either as text or as structured objects, alternating between those two views at will. The language-based editor's internal representation of a linguistic object must therefore support both local incremental parsing of arbitrarily altered text and structure-oriented editing operations[1]. Until now, editors that provide such flexibility have usually used internal representations that correspond closely to the parse tree of the concrete syntax.

Using an internal representation that is closer to the abstract syntax is preferable for several reasons:

1. The resulting representation can be chosen to correspond closely to the naive notion of the structure of the language. Matching the external structural view as anticipated by the user with the internal structure of the object as represented by the system simplifies user-interface

[1]There is a popular alternative compromise. The language being described can be partitioned between portions that can be edited textually and portions to which only structure-oriented operations apply. This approach originates with the "syntax-directed" style of editing, a style that we feel is inadequate for experienced programmers dealing with large systems.

design while preventing unpleasant surprises.

2. The resulting representation generally contains only semantically relevant structures.

3. For a given object, the resulting representation is usually much smaller than the one that corresponds closely to the parse tree of the concrete syntax.

The requirement to provide incremental parsing of textual alterations places constraints on the abstract syntax representation. These constraints would be particularly strong if we tried to parse directly from the abstract syntax. It would then be necessary to tailor the abstract syntax to the chosen parsing technology, placing limitations on the way in which semantic structures could be represented. Support for error recovery, or precedence and associativity enforcement for operators, would require modifications to the abstract syntax that conflict with the original goals of simplicity and semantic relevance.

Instead, we have chosen to relate the concrete syntax to the abstract syntax in a way that allows us to recover sufficient information from the abstract syntax representation to provide incremental parsing using the concrete grammar. We retain the advantages of a representation that corresponds closely to the abstract syntax.

Two different descriptions are required: a description of the concrete syntax is needed to parse textual representations of an object, and a description of the abstract syntax is needed to define the internal structured representation. Both of these descriptions are defined using context-free grammars. The context-free grammar that describes the textual representation is called the *concrete grammar*; the grammar that describes the abstract syntax is called the *abstract grammar*. Using separate grammars to describe the concrete syntax and the abstract syntax separates the design of the parser from the design of the internal representation.

Sometimes, a more convenient formalism to specify both the concrete and the abstract grammars may be desired. Through the use of grammar transformations, the results presented in this paper extend naturally to regular right-part grammars [15]. The details of this extension are not given here.

Although the context-free grammar formalism is useful for describing the correspondence between abstract syntax and concrete syntax, it can be awkward for describing all of the aspects of the abstract syntax representation. For that reason, we have extended our language-description notation to include additional information. Our language description actually includes three aspects: a context-free grammar that is used for parsing textual representations, a context-free grammar that serves an intermediate step between the parser grammar and the internal tree representation, and other information about the desired structure of the internal representation. In the following discussion, we describe the grammatical correspondence and the internal representation and then summarize the use of those ideas in an implementation.

1.1 Grammatical Abstraction

Grammatical abstraction is a relation between two context-free grammars. When it holds, one grammar is said to be an abstraction of the other. This form of abstraction is structural; it does not use semantic information to identify corresponding structures.

The specification of the concrete syntax, the abstract syntax, and the relationship between the two is declarative—no action routines or special procedures need be provided. Unlike other approaches in which one grammar or the other is derived automatically, the two descriptions can be modified independently. Our approach allows a high degree of control over both the structure of an internal representation and the behavior of the system during parsing. Given the two grammars, it is possible to check whether the abstraction relation holds between them. We have implemented such a checker as part of this research.

The form of abstraction we are describing is motivated by two requirements. First, it must be possible to construct an internal representation as the corresponding concrete syntax is parsed. Second, it must be possible to map from the abstract syntax back to the concrete syntax, both in order to parse changes incrementally, and in order to display the concrete syntax text to the user. In particular, the abstraction relationship assures that:

1. The abstract syntax represents a less complex version of the concrete syntax, but structures of the abstract syntax correspond to structures of the concrete syntax in a well-defined way. Both grammars describe "almost" the same formal language, subject to the renaming or erasing of symbols.

2. The relationship between the two descriptions is statically verifiable, so that developers can modify either syntax description independently.

3. The transformation from concrete to abstract is reversible, so that relevant information about a concrete derivation can be recovered from its abstract representation.

4. Efficient incremental transformations from concrete to abstract and from abstract to concrete can be be generated automatically.

1.2 Related Work

Transforming a text from an external (concrete syntax) representation to an internal (tree-structured) representation is a well-known problem in language processing. Perhaps the best known solution is the extension of syntax-directed translation called *tree-transduction grammars*[5]. This scheme uses an auxiliary tree-building stack to hold intermediate results during the translation. Each production in a context-free grammar describing the concrete syntax has an associated tree-building action. When a production is recognized, the associated action is performed.

Although the tree transduction approach handles relatively simple cases like flattening chains in expression grammars, it is restricted to being a one-to-one mapping between productions in the grammar for the concrete syntax and productions in the output (transduction) grammar. This restriction imposes undesirable limitations on the author of a language description. First, the author must specify an action explicitly for each production in the concrete grammar, including those with no semantic relevance whatsoever. Second, the description is operational, so the reversibility of the specified transformations is not guaranteed. In order to guarantee reversibility, either the set of available transformations must be constrained, or the reverse transformations must be provided explicitly. Third, the tree-transduction grammars cannot describe some cases in which a limited amount of local pattern matching (or tree-rewriting) is desirable. The grammars G_1 and $\widehat{G_1}$ of Figure 7 illustrate one such form of pattern matching where different subtrees are constructed depending on whether the "else clause" is present.

The notion of a one-to-one mapping between productions in the two grammars is too fine-grained for our purposes. A complete tree-transformation system, on the other hand, is more general than we need for incremental parsing. The notion of grammatical abstraction is somewhere in between.

The MENTOR system[6] includes a tree-building language within the specification language METAL[12]. This language can be used to perform local pattern matching as the internal tree is built. A concrete syntax is described using a context-free grammar. Associated with the concrete grammar is a specification of how to build the internal representation. The specification language permits access to the partially-built tree so that local pattern matching is possible. However, the relationship between the concrete syntax and the abstract syntax is hidden in the procedural specification of the tree-building actions, and is not available to an incremental parser. Since structure-editing is the dominant mode of user interaction supported by MENTOR, no support for fine-grained incremental parsing is provided.

The BETA project[14] also defines abstraction as a relationship between productions in two grammars. The formal definition is based on a mapping from the abstract syntax to the concrete syntax called *concrete*. This mapping is a form of unparsing by which a node in the internal representation is represented textually for display. Unlike MENTOR, there does not seem to be support for bottom-up tree building.

Both Kastens[13] and Rosenkrantz and Hunt[18] relate an abstract syntax and a concrete syntax by deriving a concrete grammar from a grammar describing the abstract syntax. The derivation (specified interactively in [13], performed automatically in the work of [18]) is captured by a tool that constructs the transformations to be used during parsing. This approach ensures the similarity of the two grammars, but does not permit the flexibility of modifying the two descriptions independently. Noonan[17] takes the opposite approach, deriving a description of an abstract syntax from the grammar used for parsing. His approach is semi-automatic; the developer must intervene in certain cases.

The work of Waddle[20] is close to our own, though developed independently. However, his intent is to reduce the size of the internal tree, called a *production tree*, by eliminating unnecessary nodes. In certain cases, the work reported here will lead to smaller trees.

Finally, the work presented here is strongly related to the notion of grammatical covers [8,16,18]. Much of the original work on covers was motivated by the same requirements encountered in this paper.

1.3 Examples

Before we present the formal definitions and results, consider the following examples[2]. Figure 1 shows a simple expression grammar G_0 with two possible abstractions. In the grammar G_0, the nonterminal symbols ⟨factor⟩ and ⟨term⟩ enforce the precedence of multiplication over addition. The grammars $\widehat{G_0}$ and $\widehat{G'_0}$ provide abstract syntax definitions for G_0 in which precedence and associativity specifications have been removed. In $\widehat{G_0}$, ⟨binop⟩ subsumes both ⟨addop⟩ and

[2] The notational conventions used in this paper are summarized below. They follow the conventions of [9].

Concrete Grammar G_0

1)	⟨expr⟩	→	⟨term⟩ ⟨addop⟩ ⟨expr⟩
2)		|	⟨term⟩
3)	⟨term⟩	→	⟨factor⟩ ⟨mulop⟩ ⟨term⟩
4)		|	⟨factor⟩
5)	⟨factor⟩	→	id
6)		|	"(" ⟨expr⟩ ")"
7)	⟨addop⟩	→	"+"
8)	⟨mulop⟩	→	"*"

Abstract Grammar $\widehat{G_0}$

1′)	⟨expr⟩	→	⟨expr⟩ ⟨binop⟩ ⟨expr⟩
2′)		|	id
3′)	⟨binop⟩	→	"+"
4′)		|	"*"

Abstract Grammar $\widehat{G'_0}$

1″)	⟨expr⟩	→	⟨expr⟩ "+" ⟨expr⟩
2″)		|	⟨expr⟩ "*" ⟨expr⟩
3″)		|	id

Figure 1: Simple Expression Grammar with Two Possible Abstractions

⟨mulop⟩. Parentheses have been eliminated from $\widehat{G_0}$, but all other terminal symbols have been retained. The grammar $\widehat{G'_0}$ provides an alternative abstract grammar for G_0 in which the nonterminal ⟨binop⟩ has also been eliminated.

Figures 2–4 illustrate derivation trees for the expression $[\![\mathsf{id}_1 + \mathsf{id}_2 * \mathsf{id}_3]\!]$ using the grammars of Figure 1. In the tree of Figure 3, the chain reductions '⟨expr⟩ → ⟨term⟩' and '⟨term⟩ → ⟨factor⟩' do not appear and the nonterminals ⟨addop⟩ and ⟨mulop⟩ have been coalesced. The tree in Figure 4 is further compacted. It is easy to see, however, that the subtree rooted at node E_2 in the concrete syntax tree (Figure 2) corresponds to the subtree rooted at E_2 in the simpler trees. The roots of the trees also correspond. Clearly, the context-free grammar $\widehat{G_0}$ is some sort of abstraction of G_0, and $\widehat{G'_0}$ is some sort of abstraction of both $\widehat{G_0}$ and G_0. The definitions that follow

Context-Free Grammar:	$G = (V, \Sigma, P, S)$
Terminal symbols:	$a, b, c, \ldots \in \Sigma$ or symbol
Nonterminal symbols:	$A, B, C, \ldots \in N \subseteq V$ or ⟨symbol⟩
Arbitrary symbols:	$X, Y, Z, \ldots \in V$
Terminal strings:	$u, v, w, \ldots \in \Sigma^*$
Arbitrary strings:	$\alpha, \beta, \gamma, \ldots \in V^*$
Empty string:	ϵ
Abstract grammar symbols:	$\widehat{X}, \widehat{\alpha}, \ldots$
Syntactic quotes:	$[\![\ldots]\!]$

The nonterminal symbol on the left-hand side of the first production of a grammar is taken to be the start symbol. Thus, in Figure 1, ⟨expr⟩ is the start symbol for G_0, $\widehat{G_0}$, and $\widehat{G'_0}$.

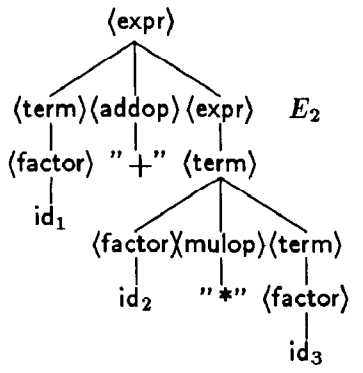

Figure 2: Tree for $[\![\mathsf{id}_1 + \mathsf{id}_2 * \mathsf{id}_3]\!]$ in G_0

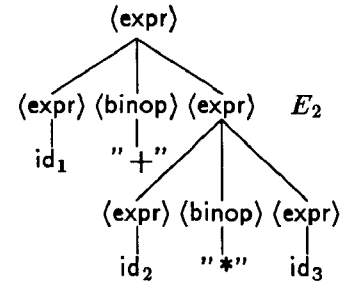

Figure 3: Tree for $[\![\mathsf{id}_1 + \mathsf{id}_2 * \mathsf{id}_3]\!]$ in $\widehat{G_0}$

capture this notion.

However, it is the derivation tree of Figure 5 that may be desired as an internal representation. In that tree, several extra ⟨expr⟩ nodes have been removed. It is possible to define such an abstract syntax by using the context-free grammar of Figure 6, but the difficulty of writing a correct specification for the abstract syntax using this style is considerable. An alternative specification method is discussed in Section 4.

Figure 7 describes the syntax of conditional statements using grammar fragments G_1, $\widehat{G_1}$, and $\widehat{G'_1}$. The two different forms of conditional statement are represented by distinct operators in the internal rep-

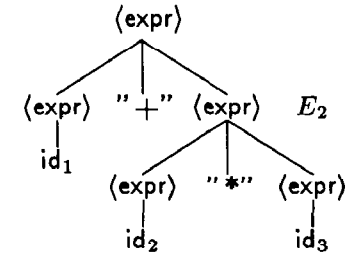

Figure 4: Tree for $[\![\mathsf{id}_1 + \mathsf{id}_2 * \mathsf{id}_3]\!]$ in $\widehat{G'_0}$

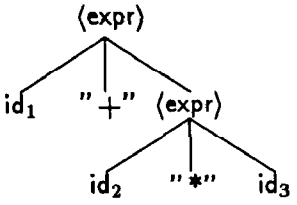

Figure 5: Desired Syntax Tree

1)	⟨expr⟩	→	⟨expr⟩ "+" ⟨expr⟩
1.1)		|	id "+" ⟨expr⟩
1.2)		|	⟨expr⟩ "+" id
1.3)		|	id "+" id
2)		|	⟨expr⟩ "*" ⟨expr⟩
2.1)		|	id "*" ⟨expr⟩
2.2)		|	⟨expr⟩ "*" id
2.3)		|	id "*" id

Figure 6: Grammar For Desired Internal Tree

resentation: those having an "else" clause, and those without an "else" clause. These examples are used to illustrate points in the remainder of the paper.

2 Definitions

Suppose $G = (V, \Sigma, P, S)$ and $\widehat{G} = (\widehat{V}, \widehat{\Sigma}, \widehat{P}, \widehat{S})$ are two reduced context-free grammars, and that $\widehat{G}$ is to be an abstraction of G. Abstraction relates a subset of the productions in a concrete grammar to each production in the abstract grammar. This is similar to the notion of grammatical covers[8,10,16,18], but the starting point differs.

Motivating the definition of abstraction are four

Concrete Grammar G_1

1)	⟨stmt⟩	→	⟨if-stmt⟩ ";"
2)	⟨if-stmt⟩	→	⟨if-part⟩ ⟨else-part⟩
3)	⟨if-part⟩	→	if ⟨expr⟩ then ⟨stmts⟩
4)	⟨else-part⟩	→	ϵ
5)		|	else ⟨stmts⟩

Abstract Grammar $\widehat{G_1}$

1′)	⟨stmt⟩	→	if ⟨expr⟩ then ⟨stmts⟩ ";"
2′)		|	if ⟨expr⟩ then ⟨stmts⟩ else ⟨stmts⟩ ";"

Abstract Grammar $\widehat{G'_1}$

1″)	⟨stmt⟩	→	⟨expr⟩ ⟨stmts⟩
2″)		|	⟨expr⟩ ⟨stmts⟩ ⟨stmts⟩

Figure 7: Conditional Statement Grammars

ideas:

1. There is a correspondence between a set of *significant* nonterminal symbols in G and the nonterminal symbols in $\widehat{G}$. Intuitively, if the significant symbol $A \in N$ corresponds to the nonterminal $\widehat{A} \in \widehat{N}$, then the two symbols represent corresponding structures in the two languages $L(G)$ and $L(\widehat{G})$. Nonterminal symbols are significant independent of context.

2. Every terminal symbol in G either corresponds to a terminal symbol in $\widehat{G}$ or is absent from $\widehat{G}$.

3. Every derivation in G has a corresponding derivation in $\widehat{G}$. Certain derivations from significant symbols in G are directly related to individual productions in $\widehat{G}$.

4. Each production in $\widehat{G}$ corresponds to a derivation in G beginning with a significant nonterminal symbol. That is, if the production $\widehat{A} \to \widehat{\alpha}$ is in $\widehat{P}$ then $A \stackrel{*}{\Rightarrow} \alpha$ in G where the significant nonterminal A corresponds to $\widehat{A}$ and the string α corresponds to $\widehat{\alpha}$.

The definition of abstraction is based upon a set of interesting nonterminal symbols $H \subseteq N$, together with a function θ_0 mapping symbols in $\Sigma \cup H$ to symbols in $\widehat{\Sigma} \cup \widehat{N} \cup \{\epsilon\}$. From the set of interesting nonterminal symbols and the initial mapping provided by the language designer, the set of significant nonterminal symbols and the extended mapping function θ are obtained. Varying H and θ_0 leads to different abstractions of the same concrete grammar.

Unlike covers, the extended mapping function θ is allowed to erase as well as rename symbols. We require that θ_0 be a function such that $\theta_0(\Sigma) \subseteq \widehat{\Sigma} \cup \{\epsilon\}$ and $\theta_0(H) \subseteq \widehat{N}$. In this case, we say that θ_0 is an *admissible abstraction map*.

As usual, we extend θ_0 homomorphically to $\theta_0 : (\Sigma \cup H)^* \to (\widehat{V} \cup \{\epsilon\})^*$ by

$$\theta_0(\alpha\beta) = \theta_0(\alpha)\theta_0(\beta) \quad \text{and} \quad \theta_0(\epsilon) = \epsilon.$$

The requirement that θ_0 be a function independent of context can be relaxed, in the case of LR(k) grammars, by using a construction similar to one given in [7].

One can think of the interesting nonterminals H as a shorthand description for the structures of G that are retained in $\widehat{G}$. Depending on the actual grammar G, the set of interesting nonterminals must be extended to include other nonterminals that represent the same abstract structures as those described

by nonterminals in H. The resulting extended set of symbols is termed *significant*.

The circularities arising from precedence enforcement are typical sources of *identified* symbols. For instance, the symbols ⟨factor⟩ and ⟨term⟩ of the expression grammar G_0 are identified with ⟨expr⟩ whenever parentheses are eliminated from the abstract syntax. The identified symbols, together with the interesting symbols constitute the set of significant symbols.

Definition 1 Assume that $G = (V, \Sigma, P, S)$ and $\widehat{G} = (\widehat{V}, \widehat{\Sigma}, \widehat{P}, \widehat{S})$ are reduced context-free grammars. Let $H \subseteq N$ be the set of interesting nonterminals of G, and let $\theta_0 : \Sigma \cup H \to \widehat{V} \cup \{\epsilon\}$ be an admissible abstraction map. Let $P_{N-H} = \{ A \to \alpha \mid A \in N - H \}$. Then *Identified* $= \{ B \in N - H \mid \exists A \in H$ such that

$$A \Rightarrow \alpha \underset{P_{N-H}}{\overset{*}{\Rightarrow}} x_1 B x_2 \text{ where } \theta_0(x_1 x_2) = \epsilon \text{ and}$$

$$B \underset{P_{N-H}}{\overset{*}{\Rightarrow}} y_1 A y_2 \text{ where } \theta_0(y_1 y_2) = \epsilon \}$$

When the nonterminal B is added to the set *Identified* because it is reachable from the interesting nonterminal A under the conditions given above, we say that B *is identified with* A.

Note that $H \cap$ *Identified* $= \emptyset$ and that all *identified* symbols participate in recursion in G.

The significant symbols are defined by

$$\textit{Significant} = H \cup \textit{Identified}.$$

The remaining *nondenotative* nonterminal symbols reflect steps in a concrete derivation that are elided in the corresponding abstract derivation. By definition, *Nondenotative* $= N -$ *Significant*. In the example grammar G_1, if ⟨stmt⟩ $\in H$ and ⟨if-stmt⟩ $\notin H$, then for most choices of θ_0, ⟨if-stmt⟩ is a nondenotative symbol.

Using the definitions of identified and nondenotative symbols, we can extend the initial mapping function θ_0 to the mapping θ on $\Sigma \cup N$.

Definition 2 Assume that $G = (V, \Sigma, P, S)$ and $\widehat{G} = (\widehat{V}, \widehat{\Sigma}, \widehat{P}, \widehat{S})$ are reduced context-free grammars. Let $H \subseteq N$ be the set of interesting nonterminals of G, and let $\theta_0 : \Sigma \cup H \to \widehat{V} \cup \{\epsilon\}$ be an admissible abstraction map. Let *Identified*, *Significant*, and *Nondenotative* be as previously defined. Let "$\perp$" be any symbol not appearing in $\widehat{V}$. (The symbol "$\perp$" is used as a notational device to avoid certain sentential forms of G when considering the correspondence with $\widehat{G}$.)

We place the following constraint upon θ_0: if A and B are two different interesting symbols, and the nonterminal C is identified with both A and B, then A and B must be treated identically in the abstract syntax. Formally, if $A, B \in H, A \neq B$; C is identified with A, and C is identified with B, then $\theta_0(A) = \theta_0(B)$. Were it not for this restriction, extra mechanism would be required to map from the abstract syntax back to the concrete syntax.

We define the extended mapping function $\theta : V \to \widehat{V} \cup \{\epsilon, \perp\}$ by

$$\theta(X) = \begin{cases} \theta_0(X) & \text{if } X \in H \cup \Sigma \\ \theta_0(A) & \text{if } X \in \textit{Identified} \text{ and} \\ & \quad X \text{ is identified with } A \\ \perp & \text{if } X \in \textit{Nondenotative} \end{cases}$$

and extend θ to a homomorphism on V^* in the usual way.

Finally, we define the set of *significant productions* to be $P_{sig} = \{ A \to \alpha \mid A \in \textit{Significant} \}$ and let $P_{non} = P - P_{sig}$.

3 Weak Grammatical Abstraction

The structures defined from H and θ_0 are used to formalize the notions of weak and strong grammatical abstraction. The definitions of both weak and strong abstraction require that the concrete syntax be described by an unambiguous context-free grammar. This requirement is not overly restrictive, since we expect that the concrete syntax will be used to generate parsers. It is possible to generalize the formal treatment to context-free grammars such as those handled by *yacc*[1] in which operator-precedence directives to the parser generator substitute for structures in the context-free grammar. However, that generalization complicates the discussion without providing additional insight to the reader, so we have not included it here.

Definition 3 Let $G = (V, \Sigma, P, S)$ be an unambiguous, reduced context-free grammar with $H \subseteq N$. Let $\widehat{G} = (\widehat{V}, \widehat{\Sigma}, \widehat{P}, \widehat{S})$ be a reduced context-free grammar. Let *Significant* be as defined previously. Let θ_0 be an admissible abstraction map, with θ the extended mapping function. Then $\widehat{G}$ is an *weak* $\langle H, \theta_0 \rangle$-*abstraction* of G if

1. $S \in H$ and $\theta(S) = \widehat{S}$, and
2. For every $A \in$ *Significant*, and for every derivation in G of the form $A \Rightarrow \beta \underset{P_{non}}{\overset{*}{\Rightarrow}} \alpha$ where $\alpha \in (\Sigma \cup \textit{Significant})^*$, either $\theta(A) = \theta(\alpha)$, or the production $\theta(A) \to \theta(\alpha)$ appears in $\widehat{P}$.

The definition of weak abstraction allows nondenotative symbols to be recursive. However, it is the case that when $\widehat{G}$ is a weak $\langle H, \theta_0 \rangle$-abstraction of G, if $A \in \mathit{Nondenotative}$ and $A \underset{P_{non}}{\overset{*}{\Rightarrow}} x_1 A x_2$ where $x_1, x_2 \in (\Sigma \cup \mathit{Significant})^*$, then $\theta(x_1 x_2) = \epsilon$. This means that circularities involving only nondenotative symbols cannot add any symbols that are represented in the abstract syntax.

Consider again the grammars G_0 and $\widehat{G_0}$ of Figure 1. The grammar $\widehat{G_0}$ is a weak $\langle H, \theta_0 \rangle$-abstraction of G_0 when $H = \{ \langle \mathsf{expr} \rangle, \langle \mathsf{addop} \rangle, \langle \mathsf{mulop} \rangle \}$ and the initial mapping θ_0 is given by

$$\theta_0(X) = \begin{cases} X & \text{if } X \in \{ \langle \mathsf{expr} \rangle, \mathsf{id}, \text{"+"}, \text{"*"} \} \\ \langle \mathsf{binop} \rangle & \text{if } X \in \{ \langle \mathsf{addop} \rangle, \langle \mathsf{mulop} \rangle \} \\ \epsilon & \text{if } X \in \{ \text{"("}, \text{")"} \}. \end{cases}$$

In this case, $\mathit{Identified} = \{ \langle \mathsf{term} \rangle, \langle \mathsf{factor} \rangle \}$ and $\mathit{Nondenotative} = \emptyset$. The extended mapping θ is given by

$$\theta(X) = \begin{cases} X & \text{if } X \in \{ \mathsf{id}, \text{"+"}, \text{"*"} \} \\ \langle \mathsf{binop} \rangle & \text{if } X \in \{ \langle \mathsf{addop} \rangle, \langle \mathsf{mulop} \rangle \} \\ \langle \mathsf{expr} \rangle & \text{if } X \in \{ \langle \mathsf{expr} \rangle, \langle \mathsf{term} \rangle, \langle \mathsf{factor} \rangle \} \\ \epsilon & \text{if } X \in \{ \text{"("}, \text{")"} \}. \end{cases}$$

If the parentheses were not erased, the grammar $\widehat{G_0}$ would *not* be a weak abstraction of G_0 given the above definition of H.

The grammar $\widehat{G'_0}$ is a weak $\langle H, \theta_0 \rangle$-abstraction of G_0 when $H = \{ \langle \mathsf{expr} \rangle \}$ and θ_0 is given by

$$\theta_0(X) = \begin{cases} X & \text{if } X \in \{ \langle \mathsf{expr} \rangle, \mathsf{id}, \text{"+"}, \text{"*"} \} \\ \epsilon & \text{if } X \in \{ \text{"("}, \text{")"} \}. \end{cases}$$

The nonterminal symbols $\langle \mathsf{addop} \rangle$ and $\langle \mathsf{mulop} \rangle$ are nondenotative symbols in this example.

Both of the grammars $\widehat{G_1}$ and $\widehat{G'_1}$ in Figure 7 are weak abstractions of G_1 when $H = \{ \langle \mathsf{stmt} \rangle \}$. For $\widehat{G_1}$, all terminals symbols shown are preserved, while in $\widehat{G'_1}$, the terminal symbols are erased. (The example does not describe the structure of $\langle \mathsf{exprs} \rangle$ or $\langle \mathsf{stmts} \rangle$.)

Theorem 1 shows that weak grammatical abstraction preserves the language and its structure up to the differences induced by the mapping θ. That is, if $\widehat{G}$ is a weak $\langle H, \theta_0 \rangle$-abstraction of G then $\theta(L(G)) \subseteq L(\widehat{G})$. Furthermore, every sentence in $L(G)$ has a structurally similar representative in $L(\widehat{G})$.

Theorem 1 Let $G = (V, \Sigma, P, S)$ be an unambiguous, reduced context-free grammar, and let $\widehat{G} = (\widehat{V}, \widehat{\Sigma}, \widehat{P}, \widehat{S})$ be a reduced context-free grammar. Let *Significant* be as defined previously and let θ be the extended mapping function of θ_0.

If $\widehat{G}$ is a weak $\langle H, \theta_0 \rangle$-abstraction of G, then for all $A \in H$, if $A \underset{G}{\overset{*}{\Rightarrow}} \alpha$ where $\alpha \in (\Sigma \cup \mathit{Significant})^*$ then $\theta(A) \underset{\widehat{G}}{\overset{*}{\Rightarrow}} \theta(\alpha)$.

Proof The details may be found in [2].

Corollary 1 If $\widehat{G}$ is a weak $\langle H, \theta_0 \rangle$-abstraction of G, then $\theta(L(G)) \subseteq L(\widehat{G})$.

Proof Follows from Theorem 1 and the fact that $S \in H$ and $\theta(S) = \widehat{S}$.

4 Building the Internal Representation

Suppose that $\widehat{G}$ is a weak $\langle H, \theta_0 \rangle$-abstraction of G where $\widehat{G}$ describes the abstract syntax and G describes the concrete syntax of a language. The definition of weak $\langle H, \theta_0 \rangle$-abstraction establishes a set of pairs $\langle A \Rightarrow \beta \underset{P_{non}}{\overset{*}{\Rightarrow}} \alpha, \widehat{A} \rightarrow \widehat{\alpha} \rangle$ that associate derivations in G with productions in $\widehat{G}$. The task of the tree-building component of a language-processing system is to recognize when the derivations in G are sufficiently complete that the subtree representing that derivation can be constructed. The tree-builder also implements the mapping θ.

Using the abstract grammar to define an internal representation has limitations in practice. The abstract grammar may include chain productions that need not be represented structurally in the internal representation. We allow the author of a language description to indicate whether a chain production in the abstract grammar should be represented structurally, by an annotation on an existing subtree, or completely omitted. This treatment of chain productions is simply an encoding of singleton subtrees. The mapping θ may induce certain chain derivations in the abstract syntax. This issue is discussed in Section 5.

During bottom-up parsing of the concrete syntax, a derivation corresponding to a production in the abstract grammar is always completed by the reduction of a production in P_{sig}. At this point in the parse, a new node in the internal representation can be created. Building the internal representation during bottom-up parsing requires an additional stack to hold subtrees until they are integrated into larger trees. When a tree node is to be created, its subtrees will be found on the tree-building stack. This stack may not parallel the parse stack, since intermediate reductions can change the parse stack without changing the tree-building stack.

There might exist two forms $\alpha_1, \alpha_2 \in (\Sigma \cup \mathit{Significant})^*$ such that $\theta(\alpha_1) \neq \theta(\alpha_2)$

and both $A \Rightarrow \beta \overset{*}{\underset{P_{non}}{\Rightarrow}} \alpha_1$ and $A \Rightarrow \beta \overset{*}{\underset{P_{non}}{\Rightarrow}} \alpha_2$ are induced by weak $\langle H, \theta_0 \rangle$-equivalence. In this case, we want to represent the two different derivations distinctly in the internal representation since they represent different semantic constructs. The reduction of the production $A \rightarrow \beta$ will not provide enough information to construct the tree properly.

Two different methods have been investigated for selecting the correct production in the abstract grammar to use when a significant production is being reduced. The first approach modifies the concrete grammar so that each production in P_{sig} corresponds to a single production in the abstract grammar. This approach can be seen as a way to merge the choice of abstract productions into the state machine of the parser. The second approach moves the mechanism out of the parser by encapsulating derivation sequences using values, called "yield-states", that are assigned to reductions in the parse. During a reduction action, a new yield-state is computed from the states of the right-hand side elements on the parse stack. In effect, the second approach introduces a separate state machine for the abstraction process.

Figure 8 contains several examples of this situation. (For simplicity, the structure of expressions is not shown.) Assume that $H = \{ \langle \text{stmts} \rangle, \langle \text{stmt} \rangle, \langle \text{expr} \rangle \}$, and $\theta_0(X) = X$ for all elements of H and Σ. Then *Significant* $= H$ and P_{sig} consists of the productions numbered 1, 2, 3, 4, and 13. Some of the correspondences between productions in the concrete grammar and those in the abstract grammars of Figure 8 are:

Concrete	Abstract
3	$3', 4'$
4	$5', 6', 7', 8'$

In each case, when the reduction of a rule in the concrete grammar occurs, the tree-building action corresponding to the appropriate production in the abstract syntax must be invoked.

4.1 Grammar Modification

Grammar modification alters the concrete grammar so that a production in P_{sig} relates to only one production in the abstract grammar. Modifying the grammar is one way to split states within the LR parser. It does not require any knowledge of the derivation sequences encountered, and can be performed either manually or automatically.

Grammar modification proceeds by finding the branches in the derivation and propagating them "backward" to the initial production. By renaming the alternative productions, duplicating productions, and propagating the differences toward the initial production, one can move the alternatives to the "top" of the derivation. This results in some number of new significant productions, each one corresponding to a single production in $\hat{P}$.

Concrete Grammar

1)	⟨stmts⟩	→	ϵ
2)		|	⟨stmts⟩ ⟨stmt⟩
3)	⟨stmt⟩	→	⟨if-stmt⟩
4)		|	⟨loop⟩
5)	⟨if-stmt⟩	→	if ⟨expr⟩ then ⟨stmts⟩ ⟨else-part⟩
6)	⟨else-part⟩	→	else ⟨stmts⟩
7)		|	ϵ
8)	⟨loop⟩	→	⟨label⟩ while ⟨expr⟩ ⟨stmts⟩ ⟨loop-tail⟩
9)	⟨loop-tail⟩	→	unless ⟨expr⟩
10)		|	repeat
11)	⟨label⟩	→	id
12)		|	ϵ
13)	⟨expr⟩	→	...

Abstract Grammar

1′)	⟨stmts⟩	→	ϵ
2′)		|	⟨stmts⟩ ⟨stmt⟩
3′)	⟨stmt⟩	→	if ⟨expr⟩ then ⟨stmts⟩
4′)		|	if ⟨expr⟩ then ⟨stmts⟩ else ⟨stmts⟩
5′)		|	while ⟨expr⟩ ⟨stmts⟩ repeat
6′)		|	while ⟨expr⟩ ⟨stmts⟩ unless ⟨expr⟩
7′)		|	id while ⟨expr⟩ ⟨stmts⟩ repeat
8′)		|	id while ⟨expr⟩ ⟨stmts⟩ unless ⟨expr⟩
9′)	⟨expr⟩	→	...

Figure 8: Grammars Showing Alternative Productions in the Abstract Grammar

In the example of Figure 8, ⟨label⟩ and ⟨loop-tail⟩ are non-recursive, non-denotative symbols, each of which has two alternatives. The grammar modification algorithm would replace those alternatives with new nonterminals, for instance replacing ⟨label⟩ by { ⟨namedL⟩, ⟨anonL⟩ } and replacing ⟨loop-tail⟩ by { ⟨unless⟩, ⟨repeat⟩ }. Substitution into production 8 then yields the new productions

8.1)	⟨loop⟩	→	⟨anonL⟩ while ⟨expr⟩ ⟨stmts⟩ ⟨unless⟩
8.2)	⟨loop⟩	→	⟨anonL⟩ while ⟨expr⟩ ⟨stmts⟩ ⟨repeat⟩
8.3)	⟨loop⟩	→	⟨namedL⟩ while ⟨expr⟩ ⟨stmts⟩ ⟨unless⟩
8.4)	⟨loop⟩	→	⟨namedL⟩ while ⟨expr⟩ ⟨stmts⟩ ⟨repeat⟩

along with the productions

$\overline{9}$)	⟨anonL⟩	→	ϵ
$\overline{10}$)	⟨namedL⟩	→	id
$\overline{11}$)	⟨unless⟩	→	unless ⟨expr⟩
$\overline{12}$)	⟨repeat⟩	→	repeat

replacing productions 9, 10, 11, and 12. Productions 8.1-8.4 can then be substituted in production 4.

The cost of altering G is the expansion of the grammar. Clearly, the transformations can be performed automatically. However, the size of the parse tables will increase.

4.2 Using Yield-States

Rather than folding rule-disambiguation into the parser, the second approach summarizes derivation sequences using state values. An additional piece of information, called the *yield-state*, is calculated as an attribute of each nondenotative nonterminal in the concrete grammar. The yield-state distinguishes among the various derivation sequences represented by that nonterminal. Yield-states are only needed when the right-hand side of a production in P_{non} contains nonterminals in P_{non} that have alternatives, or when the left-hand side has alternatives.

The yield-state value is kept on an auxiliary stack that is parallel to the parse stack. When a production in the concrete grammar is reduced, a new state value for that production is calculated. The value of the new state depends only on the rule being reduced and the values of the states associated with nonterminal symbols on the right-hand side.

For productions in P_{non}, the yield-state is computed from the yield-states of the nonterminal symbols on the right-hand side of the production. For productions in P_{sig}, the particular production in the abstract grammar to use will be defined by the yield-states of the nonterminal symbols on the right-hand side of the production in the concrete grammar.

The functions for computing yield-states and for selecting productions in the abstract grammar can be computed at language-definition time. The results can be embedded in tables that are consulted during reduce actions in the parser. An algorithm for creating the yield-state functions is given in [2].

5 Incremental Parsing

The definition of weak $\langle H, \theta_0 \rangle$-abstraction provides the foundation for adapting the incremental parsing algorithm of Jalili and Gallier [11] to use an internal representation that corresponds closely to the abstract syntax rather than the concrete syntax. Two modifications to the algorithm are required. The modifications are similar to those required to modify other incremental parsing algorithms.

A full parse tree provides three items of information at each subtree: the state of the parser, the symbol representing the root of that subtree, and the children of the node (if any) as represented in the concrete syntax. This is not the case with an arbitrary internal representation. However, when the abstract syntax is described by a grammar that is a weak $\langle H, \theta_0 \rangle$-abstraction of the concrete grammar, it is possible to associate with each node of the internal representation (i) the state of the parser when that node was created, (ii) the nonterminal symbol that represents the node in the concrete grammar, and (iii) an expansion template for mapping the node back to its representation in the concrete syntax.

5.1 Adaptation of the Jalili-Gallier Algorithm

Incremental LR parsing requires that two subproblems be solved. First, the incremental parser must be able to recreate its state at a point just prior to the first parsing action that might change because of an editing action. Second, the parser must be able to determine which portions of the parse tree have not changed despite the editing action.

One way to recreate the state information is to place, in each interior tree node, the parser state that was on top of the state stack (after popping the symbols on the right-hand side of the production) when the node was created. This is the state that was consulted by the **goto** function to compute the new state following the reduction. The initial state of the parser is placed in the node representing the root of the entire tree. This state information is used to reconstruct the parser's stack and to help determine whether a subtree can be reused without reparsing.

For an LR(1) parser, the new parse is identical to the original parse up to the point at which the last symbol in the unchanged prefix was shifted. (With a single symbol of look-ahead[3], this is the first point where any symbol in the modified string can affect a parsing action.)

Suppose that the parser's stack is to be reconstructed to the point just prior to when the terminal symbol n_{leaf} was shifted. The *Divide* operation reconstructs the parser's state stack. *Divide* splits the tree along a path from the root to n_{leaf}. The parser states associated with nodes to the left of the path (terminals and nonterminals alike) are pushed onto the growing parse stack. Nodes to the right of the path, representing potentially unchanged subtrees, are retained for later input to the algorithm. Figure 9 illustrates the *Divide* operation.

The Jalili-Gallier incremental parsing algorithm requires two modifications in order to support incremental parsing using an internal representation

[3] LR(1) or LALR(1) parsing is assumed throughout this discussion. The extension to k symbols of lookahead is straightforward.

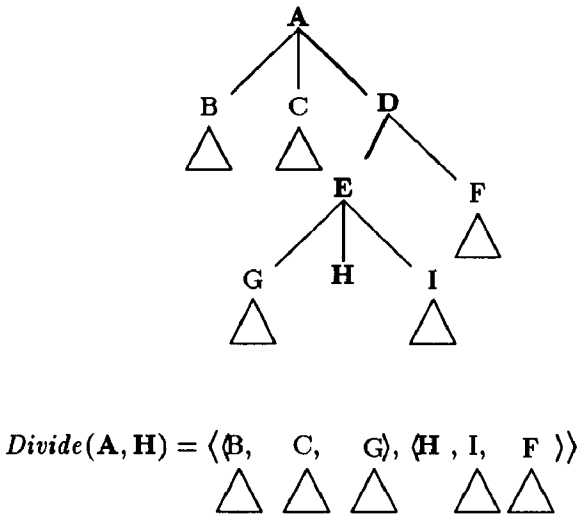

Figure 9: *Divide*(A,H)

closely related to the abstract syntax. First, *Divide* must be changed so that at each step on the path from the root to the leaf, the subtrees are expanded to their concrete representations. Second, while reconstructing the parse stack, the parse tables must be consulted to calculate the states that were not stored in the tree.

Suppose $\widehat{G}$ is an $\langle H, \theta_0 \rangle$-abstraction of G where $\widehat{G}$ describes the abstract syntax and G describes the concrete syntax. If $\widehat{A} \rightarrow \widehat{\alpha}$ is used in the internal representation created during parsing, then for some $A \in$ *Significant* and some $\alpha \in (\Sigma \cup \mathit{Significant})^*$, where $\theta(A) = \widehat{A}$ and $\theta(\alpha) = \widehat{\alpha}$, there is a corresponding derivation $A \Rightarrow \beta \underset{P_{non}}{\overset{*}{\Rightarrow}} \alpha$ in G. When a subtree whose root node represents the production $\widehat{A} \rightarrow \widehat{\alpha}$ is processed by the incremental parser, either the subtree is divided using the modified *Divide* operation, or the entire subtree can be retained. If the entire subtree is retained, then A should be the nonterminal symbol shifted by the parser.

When a subtree is divided by one of the subtree operations, it is a representation α in the concrete syntax that must actually be processed. This representation α is a single-level expansion template. Expanding $\widehat{\alpha}$ back into α generates symbols in the concrete grammar that may have been erased by the time that subtree was created. The expansion α must be determined from $\widehat{A} \rightarrow \widehat{\alpha}$ and A.

For instance, suppose the subtree representing the conditional statement '⟨stmt⟩ → ⟨expr⟩ ⟨stmts⟩' is expanded back into its concrete representation using the grammars $\widehat{G'_1}$ and G_1 of Figure 7. The expansion in the concrete syntax would be 'if ⟨expr⟩ then ⟨stmts⟩ ";"'.

Starting from the parse state that was at the top of the parse stack when the production $A \rightarrow \beta$ was reduced, the symbols in the expansion must be parsed to recover the intermediate state not present in the abstract tree. This processing will not encounter any errors, since it is the reconstruction of a correct parse. No new subtrees will be created at this stage.

For most subtrees $\widehat{A} \rightarrow \widehat{\alpha}$ in the internal representation, the associated nonterminal symbol in the concrete grammar A and the template α can be stored in tables describing the abstraction correspondence rather than in the internal representation tree. These tables can be created at parser generation time; the size of the tables depends on the language and the correspondence, not on the size of the actual internal representation. The exception occurs when the abstraction creates induced chain derivations, or when a single rule in the abstract grammar corresponds to more than one rule in the concrete grammar.

5.2 Induced Chain Derivations

Derivations of the form $A \Rightarrow \beta \underset{P_{non}}{\overset{*}{\Rightarrow}} \alpha$ where $A \in$ *Significant*, $\alpha \in (\Sigma \cup \mathit{Significant})^*$ and $\theta(A) = \theta(\alpha)$, need not be represented structurally. When $|\alpha| = 1$, the absence of a structural representation is not a problem for incremental parsing. Simple chains neither add terminal symbols to the concrete representation, nor do they have any effect on the calculation of parse states. (The parser state to be retained in the subtree is the same for all elements of a simple chain.)

When $\theta(A) = \theta(\alpha)$ and $|\alpha| > 1$, however, there is extra information to be retained; terminal symbols of the concrete grammar may have to be added to the expansion. These derivations are called *induced chain derivations*. In an induced chain derivation, a single symbol in the abstract grammar appears on the right-hand side of the derivation after the mapping θ is applied.

For instance, consider the LR parse for ⟦(id * id)⟧ using the simple expression grammar of Figure 1 when parentheses are abstracted. The abstract syntax is simply '⟨expr⟩ → id "*" id' in $\widehat{G'_0}$, but the correct expansion for incremental parsing may have to include the parentheses.

If an induced chain derivation is not represented directly by a subtree of the internal representation, the node that represents the nonterminal $\widehat{A}$ in the abstract grammar must be tagged with an expansion template for generating the terminal symbols of the induced chain derivation in order to generate a cor-

rect expansion in the concrete syntax. If the concrete representation provided by the user is to be recreated, there may be a sequence of annotations on a single node in the internal representation; the templates must be applied in the reverse order of their recognition. For instance, a node representing ⟦((((id * id))))⟧ in the internal representation would have four induced chain annotations associated with it if all four levels of parentheses are preserved.

During incremental parsing, if a subtree annotated with induced chains is retained, the nonterminal symbol used to represent the subtree should be the left-hand side of the first induced chain derivation in the sequence of annotations. In our running example, the nonterminal symbol used would be ⟨factor⟩, which corresponds to the production ⟨factor⟩ → "(" ⟨expr⟩ ")". When the subtree is divided, the subtree is expanded to restore the erased symbols. The concrete nonterminal symbol corresponding to the sole child in the internal representation should then be used to represent the child. This nonterminal symbol would be ⟨term⟩ in our example (corresponding to the production '⟨term⟩ → ⟨factor⟩ ⟨mulop⟩ ⟨term⟩').

A node in the internal representation may have n induced chain annotations associated with it. Using the straightforward approach, that node would have $n + 1$ parse states associated with it—one per induced chain reduction plus one for the original significant production. If the entire sequence of induced chain derivations is expanded during conversion from abstract to concrete then the node requires only a single associated parse state—the state corresponding to the last-recognized induced chain derivation.

5.3 Requirements on the Tree Building Component

The tree builder, then, must store or reconstruct information about each node of the internal representation in order to support incremental parsing. This information consists of the state of the parser when the subtree rooted at that node was created, the nonterminal symbol of the concrete grammar to be used to represent the node during incremental parsing, and the expansion template to be used during incremental parsing. It is possible to access both of the latter items through a single index into tables describing the weak abstraction correspondences.

During parsing, only reductions of productions in P_{sig} are of interest to the subtree creation routines. Suppose $A \rightarrow \beta \in P_{sig}$ is being reduced, and that $A \Rightarrow \beta \underset{P_{non}}{\overset{*}{\Rightarrow}} \alpha$. There are four possibilities:

1. $\theta(A) \neq \theta(\alpha)$ and $|\theta(\alpha)| > 1$. This is the normal case; a new subtree must be created and tagged with the appropriate parse state and index for recovering A and α.

2. $\theta(A) \neq \theta(\alpha)$ and $|\theta(\alpha)| = 1$. This production can be represented either structurally or by an annotation.

3. $\theta(A) = \theta(\alpha)$ and $|\alpha| = 1$. This is a simple chain rule; no tree-building actions are necessary.

4. $\theta(A) = \theta(\alpha)$ and $|\alpha| \neq 1$. This may be an induced chain derivation; if a new subtree is specified by the language designer, it will be created. Otherwise an annotation will be added to the single subtree on the right-hand side. This subtree is available from the tree-building stack. First, the annotation "$A \rightarrow \alpha$" is added to the list of induced chain annotations on the subtree, becoming the "topmost" production. This information retains both the nonterminal symbol in the concrete grammar A and the expansion template α. Second, the parser state associated with the subtree can be destructively updated to be the state that would be retained if a new subtree was being created.

6 Strong Grammatical Abstraction

Weak grammatical abstraction allows us to represent the derivation of every sentence in a concrete syntax abstractly. However, it does not allow us to map from an arbitrary sentential form generated from the abstract grammar back to corresponding concrete syntax. In order to do so, there must be at least one concrete version of every sentential form of $L(\widehat{G})$. There may be many such versions; for example, the concrete representation of an abstract expression may contain arbitrarily many levels of parenthesization. In addition, we must be certain that we can always fill in structural information missing from the abstract syntax such as precedence provided by parentheses.

One can always map back to concrete syntax, provided that (i) each nonterminal in $\widehat{G}$ maps back to a unique significant nonterminal in G and (ii) productions in $\widehat{P}$ correspond to derivations in G. The erased terminals and missing structure represented in the concrete grammar by nondenotative nonterminal symbols can then be reconstructed.

Strong (H, θ_0)-abstraction ensures that the two grammars derive the same language under θ, and provides some of the extra information necessary to implement template-based structure editing in an editor that uses bottom-up parsing.

Definition 4 Let $G = (V, \Sigma, P, S)$ be an unambiguous, reduced context-free grammar, and let $\widehat{G} = (\widehat{V}, \widehat{\Sigma}, \widehat{P}, \widehat{S})$ be a reduced context-free grammar. Let *Significant* be as defined previously, and let θ be the extended mapping function. Then $\widehat{G}$ is an *strong* $\langle H, \theta_0 \rangle$-*abstraction* of G if

1. $\widehat{G}$ is a weak $\langle H, \theta_0 \rangle$-abstraction of G.
2. The mapping θ restricted to H is invertible, that is, if $\theta(A) = \theta(B)$ and $A, B \in H$, then $A = B$.
3. For every production $\widehat{A} \rightarrow \widehat{\alpha} \in \widehat{P}$, there exists $A \in$ *Significant* such that
   (a) $\theta(A) = \widehat{A}$ and
   (b) There exists a finite derivation $A \Rightarrow \beta \overset{*}{\underset{Pnon}{\Rightarrow}} \alpha$ where $\alpha \in (\Sigma \cup$ *Significant*$)^*$ and $\theta(\alpha) = \widehat{\alpha}$.

   (It follows from this condition that $\theta(H) = \widehat{N}$.)

Again consider the grammars of Figure 1. The grammar $\widehat{G_0}$ cannot be a strong $\langle H, \theta_0 \rangle$-abstraction of G_0, since the mapping θ would have to be invertible on H. In this case, the nonterminal symbol ⟨binop⟩ in the abstract grammar would have to correspond to both ⟨addop⟩ and ⟨mulop⟩ of the concrete grammar. However, $\widehat{G'_0}$ is a strong $\langle H, \theta_0 \rangle$-abstraction of G_0.

Theorem 2 Let $G = (V, \Sigma, P, S)$ be an unambiguous, reduced context-free grammar and let $\widehat{G}$ be a strong $\langle H, \theta_0 \rangle$-abstraction of G. If $\widehat{A} \overset{*}{\underset{\widehat{G}}{\Rightarrow}} \widehat{\alpha}$ where $\widehat{\alpha} \in \widehat{V}^*$ then there exists some $A \in H$ and $\alpha \in (\Sigma \cup$ *Significant*$)^*$ such that $A \overset{*}{\underset{G}{\Rightarrow}} \alpha$, $\theta(A) = \widehat{A}$, and $\theta(\alpha) = \widehat{\alpha}$.

To prove Theorem 2, we use the following lemma:

Lemma 1 Let $G = (V, \Sigma, P, S)$ be an unambiguous, reduced context-free grammar and let $\widehat{G}$ be a strong $\langle H, \theta_0 \rangle$-abstraction of G. If $C, D \in N$ and $\theta(C) = \theta(D)$, then $C \overset{*}{\Rightarrow} x_1 D x_2$ where $\theta(x_1 x_2) = \epsilon$.

Proof of Lemma The proof is by analysis of three cases. The details may be found in [2].

Proof of Theorem 2 The proof is by induction on the length n of the derivation $\widehat{A} \overset{*}{\Rightarrow} \widehat{\alpha}$ in $\widehat{G}$.

The details are given in [2].

Corollary 2 If $\widehat{G}$ is a strong $\langle H, \theta_0 \rangle$-abstraction of G, then $\theta(L(G)) = L(\widehat{G})$.

7 Top-Down Tree Construction

Syntax-directed editors, such as MENTOR [6] or the Cornell Program Synthesizer [19], construct internal representations from the top down. Starting from the root, internal trees are elaborated by successively replacing unexpanded placeholders with subtrees. Syntactic correctness of the tree is maintained by allowing only legal subtree replacements.

The work presented in this paper allows a language-manipulation system to support textual manipulation by providing incremental parsing while using the abstract syntax to define the internal representation of a linguistic object. Yet there are times when a top-down, structure-oriented approach is desirable.

The problem of top-down tree construction in the system we have been describing is that of creating trees that are syntactically valid and that can support incremental parsing. This problem is harder than in the syntax-directed approach because we want the trees that are built top-down to be identical to the tree constructed from the same concrete representation using bottom-up parsing. Thus when a placeholder is expanded (in the internal representation), the concrete grammar must be consulted. In particular, the correspondences between productions of the abstract grammar and derivations in the concrete grammar established by strong $\langle H, \theta_0 \rangle$-abstraction will be used. The trees constructed top-down will be identical to those created by a bottom-up parsing of the same concrete form.

Theorem 2 states that if $\widehat{G}$ is a strong $\langle H, \theta_0 \rangle$-abstraction of G, then every derivation in $\widehat{G}$ has a corresponding derivation in G. This is the result needed to insure that each internal representation can support incremental parsing. In order to parse incrementally, each node of the internal representation must be decorated with the same information saved by the tree-building component of the bottom-up parsers: parse states, concrete nonterminal symbols, expansion templates, and the results of induced chain derivation reductions.

The nonterminal symbols and the expansion templates are the symbols and forms of the concrete grammar corresponding to the productions in the abstract grammar used to derive the tree. Placeholders in the tree (representing unexpanded subtrees) are nonterminal symbols in the abstract grammar. To expand a placeholder $\widehat{A}$ using the production $\widehat{A} \rightarrow \widehat{\alpha}$, some analysis is needed. The analysis may result in the addition of induced chain annotations to the subtree representing the desired expansion.

In our implementation, we handle this analysis by creating tables at language-definition time that tell

us, for any pair of nonterminal symbols, whether one symbol can be replaced with another. If the replacement is legal, the tables also encode whether induced chain annotations must be added. These tables can also be used in a batch mode to annotate internal representations created by other programs.

An unexpanded placeholder must have a corresponding representation in the concrete grammar to be processed during incremental parsing. The nonterminal symbol used is specified by the inverse of the mapping function θ. Suppose that we are expanding a placeholder $\widehat{A}$, and that $\theta(A) = \widehat{A}$ for some $A \in H$. If the incremental parser can shift the placeholder because its yield and its following symbol are unchanged, then the placeholder can use A as its concrete representation. However, if the context of a placeholder has been textually altered, this approach is insufficient.

Consider an unexpanded placeholder ⟨expr⟩ in an internal representation. This nonterminal symbol in the abstract grammar may have to be represented by the nonterminal symbol ⟨factor⟩ in the concrete grammar to ensure correct parsing. Using the definitions of θ and strong abstraction, we can compute information that allows the system to choose the symbol ⟨factor⟩ to represent the placeholder. In some cases, however, the top-down constructor must insert induced chain annotations in order to insure that the modification will be correctly parsed. Reference [4] provides the details of how top-down tree elaboration is implemented.

8 Implementation

Pan I[3] is a multi-lingual language-based editor being developed at the University of California, Berkeley. *Pan I* provides the kind of seamless text/structure interaction that we feel is essential to support experienced programmers dealing with large software systems. The work presented here provides the theoretical base for the implementation of parsing and tree building used in *Pan I*.

LADLE[4] is the language preprocessor for *Pan I*. The formal notions of abstraction are used to establish requisite correspondences between the abstract grammar and the concrete grammar. Annotations on the productions of the abstract grammar provide further information about the internal tree. When several productions of the abstract grammar are represented using a single operator in the abstract syntax—hence a single node in the internal representation—the annotation mechanism is used to preserve the information required for incremental parsing.

In a LADLE description, the language designer provides two views of the abstract syntax: as a context-free grammar and as a collection of operators defining the internal representation. The abstract grammar can be regarded as the set of representation rules for operators in the abstract syntax. Each operator in the abstract syntax is associated with one or more productions in the abstract grammar. Because the abstract grammar is viewed as a collection of representation rules, the productions in that grammar provide a place to embed pretty-printing directives.

LADLE performs the checking required to ensure that the abstract grammar is truly an abstraction of the concrete grammar. Extended BNF constructs are handled by converting them to simple context-free grammar productions using standard grammatical transformations. The results presented above, therefore, extend naturally to EBNF-described languages.

Some productions in the abstract grammar should not be explicitly represented as operators. We allow the writer of a language-specification to specify which productions in the abstract grammar will not be represented by operators in the internal representation. Those productions are limited to be simple chain productions, in which the right-hand side is a single symbol. This case arises often when the abstract syntax is described by a context-free grammar; it was discussed in Example 1.

The LADLE processor transforms a language description into a set of tables that are used by the *Pan I* editor to provide a front end for the given language. These tables contain enough information to parse text incrementally into an internal (tree-structured) representation or to elaborate such a tree from the top down.

Pan I uses a modified incremental parsing algorithm. After implementing the Jalili-Gallier algorithm described above, it was noted that a modification to the parsing tables, coupled with a change to the way subtrees were decomposed for reparsing, would allow us to parse incrementally without having to retain the parser state in each node of the internal representation. Not retaining the parse state further shrinks the internal representation and simplifies top-down elaboration at the cost of enlarged parse tables. (The parse tables are modified by calculating parser actions when nonterminal symbols in the concrete grammar are lookahead symbols.)

To date, language descriptions have been written for Modula-2, for ASPLE, and for LADLE's language description language. In the Modula-2 description, the concrete grammar contains 161 productions, and 82 nonterminal symbols, while the abstract grammar

contains 157 productions and 79 nonterminals. The total number of symbols appearing on the right-hand side of productions in the concrete grammar is 371, while in the abstract grammar it is 217. Only 97 of the productions in the abstract grammar are represented structurally in the internal tree; another 44 productions may appear as annotations.

Pan II, currently being designed, will build on the lessons learned from *Pan I*. In *Pan II*, we intend to extend the approaches presented here to provide more powerful mappings between the abstract syntax and the concrete representation.

References

[1] A. V. Aho and S. C. Johnson. Deterministic parsing of ambiguous grammars. *Communications of the ACM*, 18(8):441–452, August 1975.

[2] Robert A. Ballance, Jacob Butcher, and Susan L. Graham. *Incremental Parsing in the Pan Language-Based Editor.* Technical Report 88/??, Computer Science Division, UC Berkeley, March 1988.

[3] Robert A. Ballance, Michael L. Van De Vanter, and Susan L. Graham. *The Architecture of Pan I.* Technical Report 88/409, Computer Science Division, UC Berkeley, March 1988.

[4] Jacob Butcher. Master of science report. 1988. Computer Science Division - EECS, University of California, Berkeley. In preparation.

[5] F. L. DeRemer. *Compiler Construction; An Advanced Course*, chapter Transformational Grammars, pages 109–120. Springer-Verlag, New York, Heidelberg, Berlin, second edition, 1976.

[6] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. *Programming Environments Based on Structure Editors: The MENTOR Experience.* Research Report 26, INRIA, July 1980.

[7] Susan L. Graham. On bounded right context languages and grammars. *SIAM Journal on Computing*, 3(3):224–254, 1974.

[8] James N. Gray and Michael A. Harrison. On the covering and reduction problems for context-free grammars. *Journal of the ACM*, 19(4):675–698, October 1972.

[9] Michael A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, Reading, Mass., 1978.

[10] H. B. Hunt, D.J. Rosenkrantz, and T.G. Szymanski. The covering problem for linear context-free grammars. *Theoretical Computer Science*, 2:361–382, 1976.

[11] Fahimeh Jalili and Jean H. Gallier. Building friendly parsers. In *ACM Ninth Symposium on the Principles of Programming Languages*, pages 196–206, 1982.

[12] G. Kahn, B. Lang, B. Mélèse, and E. Morcos. Metal: a formalism to specify formalisms. *Science of Computer Programming*, 3:152–188, 1983.

[13] Uwe Kastens. Personal communication.

[14] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. An algebra for program fragments. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 161–170, 1985.

[15] Wilf R. LaLonde. Regular right-part grammars and their parsers. *Communications of the ACM*, 20(10):731–741, October 1977.

[16] Anton Nijholt. *Context-Free Grammars: Covers, Normal Forms, and Parsing.* Volume 93 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, Heidelberg, Berlin, 1980.

[17] Robert E. Noonan. An algorithm for generating abstract syntax trees. *Computer Languages*, 10(3/4):225–236, 1985.

[18] D. J. Rosenkrantz and H. B. Hunt. Efficient algorithms for automatic construction and compactification of parsing grammars. *ACM Transactions on Programming Languages and Systems*, 9(4), 1987.

[19] Tim Teitelbaum and Thomas Reps. The Cornell program synthesizer: a syntax-directed programming environment. *CACM*, 24(9):563–573, Sep 1981.

[20] Vance E. Waddle. *Production Trees: A Compact Representation of Parsed Programs.* Technical Report, IBM Thomas J. Watson Research Center, 1986.