Maptool – Supporting Modular Syntax Development

Basim M. Kadhim and William M. Waite

{Basim.Kadhim,William.Waite}@Colorado.EDU
University of Colorado, Boulder, CO 80309-0425, USA

Abstract. In building textual translators, implementors often distinguish between a concrete syntax and an abstract syntax. The concrete syntax describes the phrase structure of the input language and the abstract syntax describes a tree structure that can be used as the basis for performing semantic computations. Having two grammars imposes the requirement that there exist a mapping from the concrete syntax to the abstract syntax. The research presented in this paper led to a tool, called Maptool, that is designed to simplify the development of the two grammars. Maptool supports a modular approach to syntax development that mirrors the modularity found in semantic computations. This is done by allowing users to specify each of the syntaxes only partially as long as the sum of the fragments allows deduction of the complete syntaxes.

Keywords: Abstract syntax, concrete syntax, modularity, parsing grammar, syntax development, syntax mapping, tree construction

1 Introduction

The meaning of a construct in a programming language is often described in terms of the meanings of the components of that construct. A tree embodies the relationship between constructs and their components, and therefore much of the analysis that a compiler performs on a source program is conveniently expressed in the form of computations over a tree. Although the structure of this tree is related to the phrase structure of the source program text, the two are generally not identical.

Both the phrase structure and the tree structure can be defined by context-free grammars. The grammar describing the phrase structure is called the *concrete syntax*, while the grammar describing the tree structure is called the *abstract syntax*. Together, the two grammars provide the foundation for verifying the syntactic and semantic correctness of a program in the language. The desirability of distinguishing between a concrete and abstract syntax is discussed in [3, 4].

A concrete syntax can be input to a parser generator, which will then produce a parser for the language. In order for the concrete syntax to be usable by a parser generator, it must be unambiguous and typically must have either the LL(1) or LALR(1) property (depending on the parser generator being used) [12].

An abstract syntax describes the structure of trees over which computations are to be performed. While symbols and rules are included in a concrete syntax

to handle things like operator precedence and associativity, these distinctions are unimportant in describing semantic computations. It is inconvenient to force the abstract syntax to have such symbols and rules only so that it is also usable by a parser generator. Including such rules in the abstract syntax also unnecessarily increases the size of the tree. Consequently, it is desirable to allow for differences between the syntaxes by supporting a set of mappings.

Semantic computations can most often be divided into several different modules [7]. Name analysis, type analysis, and output generation are examples of common semantic computations that can be thought of as individual modules and can in many cases be implemented in isolation from one another. Each of these modules typically only specifies computations for some of the nodes in the tree. If the concrete syntax is completely specified, it should not be necessary to specify abstract tree fragments other than those involved in the current computation being considered. It is also desirable to be able to test each of these modules in isolation from one another.

For existing languages, it is very often the case that the implementor of a language would begin with a complete concrete syntax as assumed in the last paragraph. With a new language, an implementor might not be concerned with the syntactic structure of the language and might begin by describing the semantics of the language based on an abstract syntax tree. In this case, it should not be necessary to completely specify the concrete syntax, but rather only describe those fragments that resolve ambiguities present in the abstract syntax and those that add desired syntactic sugar.

With these strategies in mind, the research presented in this paper (and implemented in a tool called Maptool) addresses the following points:

1. It should not be required that both the concrete and abstract syntaxes provided in the input be complete as long as the two together specify a complete grammar.
2. There must be a statically verifiable mapping between the concrete and the abstract syntax.
3. Mappings between rules in the concrete and the abstract syntax must exist such that the abstract syntax can be designed to simplify the development of semantic computations.

Static verifiability of the mapping ensures that the syntaxes can be modified independently of one another, and that a module can be generated to construct an abstract syntax tree automatically from the syntax descriptions and a mapping specification.

The remainder of the paper begins with a discussion of related work in Sect. 2. Section 3 discusses the rules for mapping between the two syntaxes and notations for describing mappings that cannot be determined implicitly. Section 4 shows how the concrete and abstract syntaxes are made complete. The paper concludes with some notes about the implementation of Maptool and its use in the Eli Compiler Construction System [6].

2 Related Work

Rosenkrantz and Hunt [11] use the notion of symbolic equivalence to join several abstract symbols into a single concrete symbol in order to find a concrete grammar that falls into a parsable class (such as LL(1) or LR(1)). This work is consistent with other research in the area of grammatical covers [9]. It is our experience, however, that the notion of symbolic equivalence is more useful in the opposite direction: several concrete symbols form an equivalence class that is represented by a single symbol in the abstract syntax. This technique was first implemented in the Eli system by a tool called CAGT [2]. The research in this paper will also discuss a technique for dealing with cases in which the abstract syntax needs to distinguish, based on context, between constructs that have identical phrase structure.

Some of the mappings described in this paper bear strong resemblance to *Transduction Grammars* [5], however Transduction Grammars require that the same nonterminals be used in the original and the transformed versions of rules. This does not allow the grouping of symbols into symbolic equivalence classes. The work of Ballance et. al. [3] on grammatical abstraction is also closely related to the work presented here. Grammatical abstraction, however, does not allow the abstract syntax to distinguish between constructs with identical phrase structure.

We are unaware of any efforts directed towards the merging of syntax fragments in support of modular development.

3 Mapping

Through the remainder of the paper, we will assume that the user has provided a set of productions, P and P', representing fragments of the concrete and abstract syntaxes, respectively. Productions in P are all of the form $N \rightarrow x_1 \ldots x_n$, while productions in P' have either the form $N' \rightarrow x'_1 \ldots x'_n$ or $N' \hookrightarrow x'_1 \mid \ldots \mid x'_n$. The latter form represents a variadic node in the abstract syntax tree. Variadic nodes are used to represent lists of structures, where the number of those structures is determined at run-time and the syntactic glue (literal symbols) between those structures is not relevant to the semantics of the input. A symbol that precedes a $\hookrightarrow$ may not appear on the left hand side of any other production in P'.

The same symbol may appear in both P and P'. Symbols that appear on the left hand side of any production in P or P' are classified as nonterminals, while symbols that only occur on the right hand side of productions in P and P' are classified as terminal symbols. (These initial classifications are based only on the fragments provided in P and P' - they may be altered during the mapping.)

In addition to P and P', the user may provide a mapping specification that can contain two kinds of mappings: symbol mappings and rule mappings. These are the subject of the next two subsections. The last subsection discusses several mappings not implemented in Maptool and the reasons for their omission.

3.1 Symbol Mappings

Symbol mappings, or *symbolic equivalence classes*, are the means for specifying a group of symbols from P that are represented by a single symbol in P'. As an example, consider the concrete syntax fragment of Fig. 1, which describes a typical arithmetic expression language. The symbols *Expr*, *Term*, *Factor*, and *Primary* are used to indicate precedence levels of operators. Once the input is parsed and the abstract syntax tree is constructed, the structure of the tree embodies the precedence levels of the operators in the input. For most processors, the symbolic distinction between *Expr*, *Term*, *Factor*, and *Primary* becomes superfluous and even cumbersome. To solve this problem, Maptool allows users

$$
\begin{aligned}
Expr &\rightarrow Expr\ \text{'+'}\ Term && (1)\\
&\mid Expr\ \text{'-'}\ Term && (2)\\
&\mid Term && (3)\\
Term &\rightarrow Term\ \text{'*'}\ Factor && (4)\\
&\mid Term\ \text{'/'}\ Factor && (5)\\
&\mid Factor && (6)\\
Factor &\rightarrow \text{'-'}\ Factor && (7)\\
&\mid Primary && (8)\\
Primary &\rightarrow Integer && (9)\\
&\mid Identifier && (10)\\
&\mid \text{'('}\ Expr\ \text{')'} && (11)
\end{aligned}
$$

Fig. 1. Concrete Expression Syntax Fragment

to specify a group of symbols from P that are represented by a single symbol in P'. The notation $Expr \Leftrightarrow Term\ Factor\ Primary$ indicates that *Expr* is the *symbolic equivalent* in P' for *Expr*, *Term*, *Factor*, and *Primary* of P. More formally, $V' \Leftrightarrow x_1 \ldots x_n$ specifies that V' is the symbolic equivalent for $x_1, \ldots, x_n$ subject to the following restrictions:

1. $x_1, \ldots, x_n$ are symbols that only occur in productions of P.
2. $x_1, \ldots, x_n$ may not appear in more than one symbolic equivalence class.
3. $x_1, \ldots, x_n$ must all be classified as nonterminal symbols or all be classified as terminal symbols.
4. The classification of V' must "agree" with the classification of $x_1, \ldots, x_n$:
   (a) If V' appears in P, then its classification must be the same as that of $x_1, \ldots, x_n$.
   (b) If V' appears as a nonterminal in P', then $x_1, \ldots, x_n$ must also be classified as nonterminals.

(c) If V' appears as a terminal in P' or does not appear in either P or P', then V' assumes the classifications of $x_1, \ldots, x_n$. (The reason for this will be clarified by an example in Sect. 4.)

Every symbol in productions of P has a symbolic equivalent in the productions of P'. Symbols which do not appear in a symbol mapping specification are their own symbolic equivalents. Symbolic equivalents that do not appear in productions of P' will be added to the abstract syntax when the syntaxes are completed as described in Sect. 4. This ensures that any phrase derived from a symbol in the concrete syntax is represented by a tree that is rooted by its symbolic equivalent. As will be seen in the next section, many of the mappings between rules are inferred as a result of the symbolic equivalence class specifications defined in this section.

Applying a symbolic equivalence class to concrete rules means replacing each occurrence of a symbol in the production with its symbolic equivalent. In our example in Fig. 1, applying the symbolic equivalence class $Expr \Leftrightarrow Term\ Factor\ Primary$ causes rules 3, 6, and 8 to have the form $Expr \rightarrow Expr$. These are called *trivial chain productions*. They serve no purpose in the abstract syntax and do not require tree nodes to represent them. Eliminating them can result in significant reduction in the size of the abstract syntax tree.

3.2 Rule Mapping

Rule mapping provides the basis for the completion of the syntaxes described in the next section. It is also the determining factor in how the module to automatically construct the abstract syntax tree is generated. While it is possible to specify rule mappings explicitly, the rule mappings can be determined implicitly in most cases.

For each mapping, we will briefly discuss how the abstract syntax tree fragment is constructed. The tree construction is based on a scheme in which constructed subtrees are placed on a stack and used when concrete syntax rules are reduced to construct new subtrees to place on the stack. This is very much like the scheme used for tree transduction grammars [5].

Variadic Nodes Languages are replete with lists of things, such as lists of declarations or lists of statements. While parsing grammars require the introduction of recursive rules to describe such lists, it is convenient to view such a list as a single flat context when describing the semantics. This can result in a reduction in the size of the abstract syntax tree, because numerous nonterminals used in the concrete syntax to adequately disambiguate the phrase structure need not be represented in the abstract syntax tree.

As an example, consider the concrete syntax fragment in Fig. 2 representing a list of *Decl*'s followed by a list of *Stmt*'s. All of the rules in the figure can map to a variadic production in the abstract syntax of the form $Program \hookrightarrow Decl \mid Stmt$. The concrete syntax fragment includes the syntactic sugar (brackets, commas, and semicolons) and rules that enforce the requirement that *Decl* nodes precede

Stmt nodes. Neither the syntactic sugar nor the ordering need be specified in the abstract syntax. Omitting them prevents over-specification and allows one to describe semantic computations that are less specific to the exact phrase structure of the input language. In this example, the size of the abstract syntax tree is also reduced because *StmtList* and *DeclList* nodes are not represented in the tree.

$$
\begin{aligned}
\mathit{Program} &\rightarrow \text{'['}\ \mathit{DeclList}\ \text{']'}\ \mathit{StmtList} \\
\mathit{DeclList} &\rightarrow \mathit{DeclList}\ \text{','}\ \mathit{Decl} \\
&\mid \mathit{Decl} \\
\mathit{StmtList} &\rightarrow \mathit{StmtList}\ \text{';'}\ \mathit{Stmt} \\
&\mid \mathit{Stmt}
\end{aligned}
$$

Fig. 2. Concrete Representation of a Variadic Node

Variadic productions in the abstract syntax fragment P' are the first to be mapped. The process begins by examining the left hand side symbol, N', of the production. We add N' and any symbols that have N' as their symbolic equivalent to a set of symbols, R, that represents the root of the variadic context. Assume that S represents the set of symbols that appear on the right hand side of the variadic production and all symbols that have a member of S as their symbolic equivalent. Each rule in P that appears along a derivation path from a member of R to a member of S is marked as being mapped to the variadic production. More formally, the set of rules mapped to the variadic production can be described as follows:

$$\{N \rightarrow x_1 \ldots x_n \mid \exists (r \in R, s \in S) \text{ such that } r \stackrel{*}{\Rightarrow} N \stackrel{+}{\Rightarrow} s\}$$

(In the above notation, $\stackrel{*}{\Rightarrow}$ and $\stackrel{+}{\Rightarrow}$ denote the application of zero or more and one or more productions from P, respectively.)

Each non-literal symbol that appears on a derivation path from a member of R to a member of S is called an *intermediate nonterminal*. An error has occurred if any intermediate nonterminal appears in any productions of P that are not mapped to the variadic production. In other words, there can be no symbols in P from which a derivation exists to any of the intermediate nonterminals without passing through a member of R.

An error must also be signalled if a non-literal terminal symbol that is not a member of S is found on the right hand side of any production encountered on any derivation path from members of R to members of S. This would imply that an element of the input other than those specified by the variadic production was found on a derivation sequence from the root of the variadic context.

Based on this mapping, we must establish how to build the abstract syntax tree fragment representing the variadic node given the set of mapped concrete syntax rules. For each production mapped to the variadic context, the right hand side symbols that are members of S have subtree fragments on the tree construction stack. The reduction of a mapped rule must append these fragments to a running list. Every concrete production that has a member of R on its right hand side and is not mapped to the variadic production must finalize the list to create the variadic node. The reason for doing the finalization in the parent context is that it is possible for a member of R to be recursively defined. In such a case, trying to finalize the variadic node in a context that defines a member of R might execute the finalization prematurely.

Implicit Rule Mapping We say that variadic productions are mapped *implicitly* to a set of rules in the concrete syntax fragment, because the user provides no additional information to Maptool to deduce the mapping. The more general form of implicit rule mapping simply maps rules that have identical signatures. Only mapping rules with identical signatures, however, does not allow users to semantically distinguish between constructs that are syntactically the same. Fig. 3 shows two concrete rules that represent different contexts for the terminal symbol *Identifier*. Semantically, one represents the definition of an identifier and the other represents the use of an identifier.

$$Decl \rightarrow Type\ Identifier$$
$$AssignStmt \rightarrow Identifier\ '='\ Expression$$

Fig. 3. Concrete Syntax for Identifier Definitions and Uses

Symbolically distinguishing between these two different uses of *Identifier* may simplify semantic computations. Systems (like Eli [6]) where computations can be provided for occurrences of symbols in the abstract syntax tree benefit by not having to distinguish between occurrences of the same symbol in different contexts. This technique is used heavily in developing modular and reusable attribute grammars as demonstrated in [7].

The abstract syntax fragment that distinguishes between identifier uses and definitions is shown in Fig. 4. The figure shows that two new symbols, $IdDef$ and $IdUse$, are introduced to represent definitions and uses, respectively. To retain a mapping with the concrete syntax, *chain productions* are added between the new nonterminal and *Identifier*. To facilitate such mappings, Maptool not only maps rules that have identical signatures, but also rules that differ only in the names of their right hand side non-literal symbols. In cases where these symbols differ, there must be a series of chain productions connecting the two symbols.

$$Decl \rightarrow Type\ IdDef$$
$$AssignStmt \rightarrow IdUse\ \text{'='}\ Expression$$
$$IdDef \rightarrow Identifier$$
$$IdUse \rightarrow Identifier$$

Fig. 4. Abstract Syntax for Identifier Definitions and Uses

More formally, Maptool takes rules from the concrete syntax (from P) that were not mapped to a variadic context and tries to map them to rules in P'. A rule from P is mapped to a rule in P' if the following conditions hold:

1. The rules differ only in the names of their nonterminals.
2. The left hand side nonterminal symbols of the two productions belong to the same symbolic equivalence class.
3. For each pair of right hand side symbols that differ in the two productions, they either both belong to the same equivalence class or the symbol from the production in P is *coercible* to the symbol from the production in P'.

For a symbol X to be coercible to another symbol Y, there must be a series of chain productions in P' such that one can derive X from Y using those chain productions. Chain productions are productions that have only a single nonterminal symbol on the right hand side. If more than one rule in P' satisfies the above criteria, then the mapping that requires the use of the fewest chain productions is chosen. If the map is still ambiguous, an error must be signalled to the user.

The reason that coercions are possible is that, when reducing a concrete syntax rule, we can check to see what symbols are at the roots of the tree fragments on the tree construction stack. If the symbol does not match what is expected for the particular tree construction function invoked, then the tree constructor can coerce the found symbol to the expected symbol by inserting the chain productions representing the coercion sequence between the two symbols.

Explicit Rule Mapping There are times when symbolic equivalence classes and implicit rule mapping do not provide enough flexibility. For example, DeRemer recognized the desire for *standardization* of certain language constructs [4]. The example he gives involves "let expressions" and "where expressions." "Let expressions" have the form $LetExpr \rightarrow$ 'let' $Definitions$ 'in' $Expr$, while "where expressions" have the form $WhereExpr \rightarrow Expr$ 'where' $Definitions$. The syntax is different, but the semantics of the two kinds of expressions are identical.

If we have a language that allows both, we would like to have a single abstract syntax tree structure to represent both syntactic constructs. For example, our abstract syntax might have the form $BoundExpr \rightarrow Definitions\ Expr$.

To implement this kind of transformation, we allow users to provide a mapping specification that rewrites the right hand side of certain concrete syntax rules before implicit rule mapping is performed. In this case, we would have the following specification:

$$LetExpr \rightarrow \text{'let'}\ Definitions\ \text{'in'}\ Expr\ <\$1\ \$2>$$

$$WhereExpr \rightarrow Expr\ \text{'where'}\ Definitions\ <\$2\ \$1>$$

The symbols between angle brackets specify how the rule is to be rewritten. Literal symbols may be included in the rewrite and the non-literal symbols in the rule are represented by the $ symbol followed by a number, i, that represents the i-th non-literal symbol on the right hand side of the rule. Here we have stripped the literal symbols from the rules and reordered the non-literals in the case of the where expression.

3.3 Mappings Not Implemented

Maptool has been in use for over a year and implements those syntax mappings that we have found most useful. Three other possible mappings have been suggested to us. In two cases, the mappings do not agree with our goal for modular syntax development and we have not yet found compelling reasons to implement the third.

Deducing Symbolic Equivalence Classes The first suggestion was to deduce symbolic equivalence classes. Currently, Maptool requires that an explicit specification for equivalence classes be supplied by a user.

In the example in Fig. 1, the symbols *Expr*, *Term*, *Factor*, and *Primary* are all connected by chain productions. If *Expr* is the only symbol specified in an abstract syntax fragment supplied by the user, one could deduce that all of the symbols connected to *Expr* by chain productions should belong to the same equivalence class.

The problem with this deduction arises when developing syntaxes piece by piece in a modular fashion. (It is important to note that Maptool results derived from syntax fragments under development have been very heavily used, particularly with large grammars.) Consider the concrete syntax fragment in Fig. 5 in conjunction with the expression language from Fig. 1. Together they specify the complete concrete syntax for an expression language with bound variables. In developing the abstract syntax, we might first use Maptool to deduce a complete version of the abstract syntax without supplying any abstract syntax fragment.

If we allowed Maptool to deduce symbolic equivalence classes, it would deduce an equivalence among *Computation*, *Expr*, *LetExpr*, *WhereExpr*, *Term*, *Factor*, and *Primary*, because they are all connected by chain productions. For some semantic computations, however, *Computation* should not be considered part of the equivalence class. Consequently, *Computation* would either not appear in the resulting abstract syntax or would appear in unexpected contexts

$$
\begin{aligned}
\mathit{Computation} &\rightarrow \mathit{Expr} \\
&\mid \mathit{LetExpr} \\
&\mid \mathit{WhereExpr} \\
\mathit{LetExpr} &\rightarrow \text{'let'}\ \mathit{Definitions}\ \text{'in'}\ \mathit{Expr} \\
\mathit{WhereExpr} &\rightarrow \mathit{Expr}\ \text{'where'}\ \mathit{Definitions} \\
\mathit{Definitions} &\rightarrow \mathit{Definitions}\ \text{','}\ \mathit{Definition} \\
&\mid \mathit{Definition} \\
\mathit{Definition} &\rightarrow \mathit{Identifier}\ \text{'='}\ \mathit{Expr}
\end{aligned}
$$

Fig. 5. Concrete Syntax Fragment

(depending on which of the symbols is chosen by the tool to be the symbolic equivalent).

Given the size of the example just given, such a misleading result from a mapping tool would be simple to correct by adding rules to the abstract syntax fragment. Unfortunately, the larger the syntax the more difficult it becomes to detect and correct such inconsistencies. As a result, we find it preferable to force the user to specify symbolic equivalence classes explicitly.

Adding Chain Rules Another possibility that has been suggested is to deduce similarities in the signatures of rules in the concrete and the abstract syntax, and to introduce chain rules between the differing symbols in cases where it would facilitate a match. As an example, consider the concrete and the abstract syntax fragments from Figs. 3 and 4. Given the concrete syntax fragment from Fig. 3, the suggested mapping would relieve the user from having to supply the last two rules of Fig. 4. The similarity of the first two rules of the abstract syntax fragment with the two rules in the concrete syntax fragment would be deduced by the tool and the appropriate chain rules would be automatically introduced.

The problem with this approach is that it may deduce a similarity between rules that are clearly not meant to be matched. Figure 6 shows two similar concrete syntax rules that might appear in the same grammar. Maptool aims to support a style of development in which users can develop semantic computations for *Declarations* and *Statements* separately and test those computations in isolation from one another. In such a case, suppose that the user provides computations for *Declarations* (i.e. the first rule of Fig. 6 also appears in the abstract syntax fragment). If the second rule does not also appear in the abstract syntax fragment, then the suggested mapping would introduce a chain rule between *Statements* and *Declarations*. Since we want to develop separate computations for *Statements* and *Declarations*, this is clearly not the desired effect and is likely to be misleading.

Section → 'begin' *Declarations* 'end'
Section → 'begin' *Statements* 'end'

Fig. 6. Two Similar Concrete Syntax Rules

Multi-Level Matching Given the concrete syntax rule $A \rightarrow B\ C\ D$, it has been suggested that one might want to construct two contexts in the abstract syntax. These might be of the form $A \rightarrow X\ D$ and $X \rightarrow C\ D$. Conversely, one might consider mapping two concrete syntax rules of the form $A \rightarrow X\ D$ and $X \rightarrow C\ D$ to a single abstract syntax rule of the form $A \rightarrow B\ C\ D$.

Maptool does not currently support these kinds of mappings. We have not yet found compelling examples to warrant the effort required to add them. The implementation for the first of the two mappings would be fairly straightforward. The second mapping, however, would require techniques described by Ballance et al.[3] for either grammar modification or the computation of yield states in the parser.

4 Completing the Syntaxes

After the mappings are complete, Maptool must try to generate reduced context free grammars from the fragments provided. The basic technique is that rules in P are added to the abstract syntax if they do not map to any rules in P' and rules in P' are added to the concrete syntax if there aren't any rules in P that map to them. This basic technique is subject to the following restrictions:

1. All symbolic equivalence classes are applied to rules from P before they are added to the abstract syntax.
2. As indicated in Sect. 3.1, trivial chain productions are not added to the abstract syntax.
3. *Literal chain productions* are also not added to the abstract syntax. Literal chain productions are rules that have the form, $N' \rightarrow x\ N'\ y$, where x and y are sequences of literals and x or y is non-empty. The rationale here is that such a context is only needed in the abstract syntax tree if there is actually some computation associated with it, in which case it should be included in P'.
4. A chain production $X' \rightarrow Y'$ of P' is only added to the concrete syntax if X' is a symbol that already exists in productions of the concrete syntax. If X' does not already exist in the concrete syntax, then introducing the rule would result in X' being unreachable. Such productions are often present in the abstract syntax to define coercions as described in Sect. 3.2. Note that adding the chain production may result in the addition of Y' to the concrete syntax. This means that all abstract chain productions must be checked for inclusion each time a new rule is added.

5. If no rules in P are found to map to a variadic production in P', then a canonical left-recursive version of the rule is added to the concrete syntax.

Once the basic technique has been applied, Maptool must verify that the grammars are, in fact, reduced by checking that neither has more than one potential root symbol.

In Sect. 3.1 it was indicated that symbols that appear as terminals in P' assume the nonterminal classification if they are symbolic equivalents for nonterminals in P. This is because rules defining the nonterminals from P will be added to the abstract syntax making the symbolic equivalent in P' a nonterminal.

Maptool does not attempt to "undo" symbolic equivalence classes for rules in P' that are added to the concrete syntax. The reason for this can be seen in the example in Fig. 1. Again assume that we define the symbolic equivalence class, $Expr \Leftrightarrow Term\ Factor\ Primary$. Now consider the case in which we have a rule from P' that needs to be included in P and that has $Expr$ on its right hand side. we would have to generate four instances of this rule in the concrete syntax, one for each member of the symbolic equivalence class. Doing so would make the grammar ambiguous.

5 Implementation

Maptool is described by a set of specifications. Eli uses these specifications to produce a C program, which can be compiled and executed independent of Eli. The generated program takes as input a BNF grammar specifying the concrete syntax fragment, an attribute grammar specifying the abstract syntax fragment, and a mapping specification written in a language designed for Maptool.

Given these inputs, Maptool yields a complete concrete syntax annotated with reduction actions that build the appropriate abstract syntax tree using the tree construction module exported by the LIGA attribute grammar evaluator [8]. It also produces a complete abstract syntax in notation usable as input to LIGA.

An interesting part of the Maptool implementation is its use of OIL, an operator identification library available in the Eli system. Operator identification is a well understood process, first discussed by Aho and Johnson in 1976 [1, 10]. The signatures of abstract productions can quite easily be cast as operators, chain rules are represented by coercions, and the signature of the concrete syntax rules are analogous to expressions whose operators must be determined. A cost value of 1 is associated with each coercion so that the mapping requiring the fewest chain productions is chosen.

Maptool has been a component of the Eli system for over a year and has been successful in supporting the development of modular specifications in concert with the support for modularity afforded by the LIGA evaluator [7].

6 Acknowledgments

We would like to thank the members of the Eli team, with special thanks to Dr. Uwe Kastens, for their numerous contributions to this research.

This work was partially supported by the US Army Research Office under grant DAAL03-92-G-0158.

References

1. Alfred V. Aho and Stephen C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, July 1976.
2. Ali Bahrami. CAGT – an automated approach to abstract and parsing grammars. Master's thesis, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO, 1986.
3. Robert A. Ballance, Jacob Butcher, and Susan L. Graham. Grammatical abstraction and incremental syntax analysis in a language-based editor. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 185–198. SIGPLAN, ACM, June 1988.
4. Franklin L. DeRemer. *Compiler Construction; An Advanced Course*, chapter 2.E., pages 121–145. Springer-Verlag, New York, Heidelberg, Berlin, second edition, 1976.
5. Franklin L. DeRemer. *Compiler Construction; An Advanced Course*, chapter 2.A., pages 37–56. Springer-Verlag, New York, Heidelberg, Berlin, second edition, 1976.
6. Robert W. Gray, Vincent P. Heuring, Steve P. Levi, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, February 1992.
7. U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.
8. Uwe Kastens. LIGA: A language independent generator for attribute evaluators. Technical Report Reihe Informatik 63, Universität-GH Paderborn, Fachbereich Mathematik-Informatik, 1989.
9. Anton Nijholt. *Context-Free Grammars: Covers, Normal Forms, and Parsing*, volume 93 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1980.
10. Guido Persch, Georg Winterstein, Manfred Dausmann, and Sophia Drossopoulou. Overloading in preliminary Ada. *SIGPLAN Notices*, 15(11):47–56, November 1980.
11. D. J. Rosenkrantz and H. B. Hunt. Efficient algorithms for automatic construction and compactification of parsing grammars. *Transactions on Programming Languages and Systems*, 9(4):543–566, October 1987.
12. William M. Waite and Gerhard Goos. *Compiler Construction*. Texts and Monographs in Computer Science. Springer-Verlag, New York, Berlin, Heidelberg, Tokyo, 1984.