MECHANICAL GENERATION OF CONCRETE SYNTAX RULES FOR THE SCHÜTZ SEMANTIC EDITOR

A Thesis by

Christopher Wayne Johnson

B.A., Wichita State University, 2001

Submitted to the Department of Computer Science
and the faculty of the Graduate School of
Wichita State University
in partial fulfillment of
the requirements for the degree of
Master of Science

May 2007

MECHANICAL GENERATION OF CONCRETE SYNTAX RULES FOR THE SCHÜTZ SEMANTIC EDITOR

I have examined the final copy of this thesis for form and content, and recommend that it be accepted in partial fulfillment of the requirement for the degree of Master of Science in Computer Science:

_____
Rajiv Bagai, Committee Chair

We have read this thesis
and recommend its acceptance:

_____
Li Jia, Committee Member

_____
Prakash Ramanan, Committee Member

ABSTRACT

Schütz is a programmer's editor designed by Dr. Rodney Bates. Through its language definition language, users may adapt Schütz to any programming language. Describing a programming language in the language definition language involves writing largely parallel rule sets in three different syntaxes—the concrete syntax, the abstract syntax, and the format syntax. In this thesis, we present a method for mechanically generating the concrete syntax rule set, given the rule sets of the other two syntaxes; we also prove that the concrete syntax rule sets thus generated are correct and unambiguous, given the same traits in the rule sets of the other two syntaxes.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Schütz is a programmer's editor. Programmer's editors are text editors with special facilities for editing computer programs. One such facility is syntax highlighting—color-coding parts of source code, like language keywords, user-provided identifiers, numbers, etc., making source code easier for human users to read. Schütz is also a semantic editor—an editor that provides operations that take advantage of the static semantic rules of the language of the source code being edited. These functions may be simple, such as finding the declaration of an identifier based on its point of usage; or complex, such as automatically generating that declaration. Furthermore, these functions are available even for syntactically incorrect programs; and, in fact, Schütz is able to suggest repairs for such programs. Static semantic operations are desirable because they make programming easier and less error-prone for humans. Schütz's chief developer is Dr. Rodney Bates, formerly an assistant professor of Computer Science at Wichita State University.

Semantic editors are not new—the field is at least fifteen years old. Older semantic editors tended to have significant limitations, however, that discouraged their adoption. A few such editors used a syntax-directed interface—that is, the user could not type text freely, but had to enter language constructs, such as functions and while-loops, via menus and prompts. Programmers rejected those editors because working with them was too cumbersome. Other semantic editors had free-form interfaces like a standard programmer's editor, yet could do only simple semantic analysis, and even then, only on code that was complete and correct. This limited both the utility and the potential of semantic editing with those editors. Schütz

overcomes both of these classes of problems by offering a free-form interface and by working with incomplete and/or incorrect code.

The two fundamental forms of data Schütz works with are *edited texts* and *text syntax trees*. An edited text is a unit of source code—e.g., a .c file in the C programming language. A text syntax tree (TST) [4] is an abstract syntax tree (AST) [4] adorned with additional information. To perform static semantic analysis on an edited text, Schütz must first convert that edited text into a TST. In order to show the results of a static semantic analysis, Schütz must translate that TST back into an edited text. With a typical AST, this latter conversion would be impossible; thus the need for TSTs and their additional information.

Consider this piece of source code in a toy language of algebraic expressions:

```
(A+B) * C
```

The AST for this expression might look like Figure 1. The TST would look similar to Figure 1, except each Id node would contain information about its spelling (i.e., whether it was the one spelled A, B, or C).

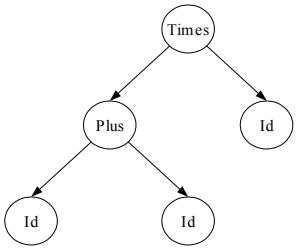

Figure 1. AST for (A+B) * C

ASTs are so called because they abstract away details in the edited text like whitespace, comments, and even particular identifier names (as shown in Figure 1). Since Schütz needs to be able to reconstruct edited text from the syntax trees, however, those details must be retained.

TSTs augment ASTs with those details (and many other details irrelevant to this thesis), helping Schütz reconstruct the edited text. Stripping those details from a TST results in an AST; Schütz uses a special program module called astview to do this. The details of this conversion are outside the scope of this thesis, so we simply assume the conversion happens as needed.

Two important components, or modules, of Schütz are the *parser* and the *formatter*. The parser converts edited texts to TSTs. The formatter converts TSTs to edited texts. This process is illustrated in Figure 2. In Figure 2, the output of the formatter is called formatted text; this is synonymous with edited text, but we name it differently in the figure to emphasize that what comes out of the formatter may not superficially match what went into the parser; the two bodies of text will be syntactically identical, but may differ in layout (i.e., in the placement and amounts of whitespace that separate tokens).

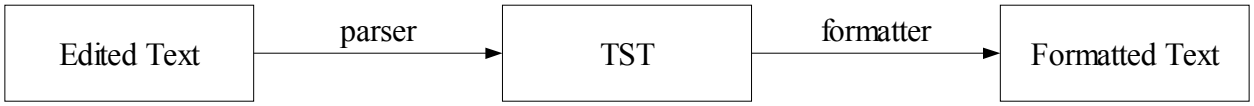

Figure 2. The parser and formatter modules.

Users can adapt Schütz to the text-based programming language of their choice by using Schütz's language definition language (LDL). LDL is, itself, a text-based programming language. A body of LDL code defining a programming language is called a *language definition*. Special tools that come with Schütz prepare language definitions for use by the parser and formatter; the details of these tools are largely out of the scope of this paper. As of this writing, Schütz uses a version of LDL called LDL1 [1]. One prior version of LDL exists—LDL0 [2]—but it was intended only to bootstrap LDL1 into Schütz [3], and is unsuitable for

general use. For this reason, whenever we use the term LDL in this paper from this point on, we refer specifically to LDL1.

A language definition in LDL has several parts: a language name, a start symbol, a precedence rule, a set of concrete syntax (CS) rules, a set of abstract syntax (AS) rules, and a set of format syntax (FS) rules. The CS rules, AS rules, and FS rules are so-called because they're written in the CS, AS, and FS, respectively. Be mindful of the difference between a syntax and a rule set in that syntax: A syntax is part of LDL itself; a rule set is part of a language definition. We illustrate the different parts of a language definition with an example capable of parsing the toy expression, (A+B) * C. Note that the numbers prefixing each line are intended for reader clarity in this example, and are not part of the language definition iteself (nor are they syntactically legal in LDL):

```
 1   LDL DemoLang
 2
 3   START Expr .
 4
 5   Id VARTERM .
 6
 7   Expr = Plus | Times | Id .
 8
 9   Times | Plus :: Left : Expr ; Right : Expr .
10
11   Plus -> _ "+" _ .
12   Times -> MEMBER Plus ( "(" _ ")" ) "*"
13             MEMBER Plus ( "(" _ ")" ) .
14
15   ExprNotPlus ::= Times | Id .
16   Plus ::= Expr "+" Expr .
17   Times ::= "(" Plus ")" "*" "(" Plus ")" .
18   Times ::= "(" Plus ")" "*" ExprNotPlus .
19   Times ::= ExprNotPlus "*" "(" Plus ")" .
20   Times ::= ExprNotPlus "*" ExprNotPlus .
21
22   END DemoLang .
```

Line 1, `LDL DemoLang`, declares that a new language definition is starting. The defined language is named DemoLang. This name is only used again at the end of the language definition, line 22's `END DemoLang .`, which declares that the new language definition is complete.

Line 3, `START Expr .`, declares `Expr` to be the start symbol of the defined language. `Expr` must thus be declared as a class or abstract node kind elsewhere in the language definition.

Line 5, `Id VARTERM .`, declares Id to be a variably spelled terminal. Such terminals may be spelled any number of ways. These spellings are defined by a language-specific scanner separate from LDL, and are not syntactically relevant. They are semantically relevant, however, and the particular spellings are stored in the TST. The mechanisms for scanning variably spelled terminals are outside the scope of this paper, so we simply assume it happens correctly.

Line 7, `Expr = Plus | Times | Id .`, declares `Expr` to be a class. A class is a set of variably spelled terminals, abstract node kinds, and other classes. Here, `Expr` = { `Plus`, `Times`, `Id` }. In the language definition, we can think of being able to substitute a class for any of its members. In our example, any other occurrence of `Expr` (besides the one in line 7) in the language definition could be substituted with `Plus`, `Times`, or `Id`. Classes function as a short-hand—they keep the language definer from having to write separate rules for each of a class' members.

Line 9, `Times | Plus :: Left : Expr ; Right : Expr .`, is an AS rule. AS rules describe the shape of TSTs (and, thus, ASTs, hence the name abstract syntax rules) the parser builds for edited texts of the defined language. This example declares two node types—`Times` and `Plus`—as having the same complement of children. Note that `Times` and `Plus` are still considered distinct node types, they're simply declared together in the language

definition for convenience. `Times` and `Plus` both have two children, named `Left` and `Right`, both of type `Expr`. Remember that `Expr` is a class. So, this one AS rule is equivalent to 18 simpler AS rules—there are two choices for left-hand side (LHS), `Times` or `Plus`; there are three choices for substitution for `Left`'s `Expr`—`Times`, `Plus`, or `Id`; and there are the same three choices for `Right`'s `Expr`. An example of one of these simpler rules would be `Plus :: Left : Plus ; Right : Plus ..`

Line 11, `Plus -> _ "+" _ .`, is an FS rule. FS rules describe how the formatter translates TSTs into edited texts. Every AS node type needs its own FS rule; this correspondence is manifest in a language definition by AS rules and FS rules having the same LHSs. In this example, we define an FS rule for the node type `Plus`. To format a `Plus` node, the formatter should format and output the first child (`Left`), then output the insertion token `+`, then format and output second child (`Right`). The underscore is a special symbol that can be thought of as being the class of all possible children. Of course, in the AS rule associated with `Plus`, each child must be of type `Expr`; this means that, in practice, the formatter will only ever emit formatted `Expr`s for those children, even though the underscore seems to suggest it could format any node kind. By "format and output" we mean the formatter will use the FS rule associated with the child type (`Expr`, in this case) to format that child; this process is recursive. Note how we recover the spelling of the operator even though it's not stored in the TST (because there's no mention of that operator in the associated AS rule).

Lines 12 and 13 describe the formatting for the `Times` node. These use `MEMBER` predicates to specify a different kind of formatting depending on the type of the child node. In

this example, if either child is a `Plus` node, the formatter outputs parentheses around it; else, the formatter simply formats and emits the child sans parentheses.

On lines 15 though 20 are declared a series of CS rules. CS rules describe how the parser should translate edited texts into TSTs. If the LHS of a CS rule has the same name as a node type, the parser will build that node type as a consequence of parsing for that CS rule. For instance, line 16, `Plus ::= Expr "+" Expr .`, declares a CS rule to build a `Plus` node. Note the correspondence between the node types in the CS rule and those in the associated AS rule. Only node types are counted in TST building, not insertion tokens like +.

Note the complexity in lines 15 and 17 through 20. In line 15, we declare what amounts to a 2/3rds class for use by other CS rules. This is necessary to describe how to recognize necessary parentheses (note that the defined language does not allow unnecessary parentheses). Lines 17 through 20 declare CS rules for parsing multiplication expressions and building `Times` nodes. In the FS rule for `Times`, the FS affords us the expressiveness (through the `MEMBER` predicates) to hide this complexity; the CS offers no such mechanism.

We can make several observations about this example. One observation is, the rule sets for the different syntaxes need to match up in the order and types of child nodes. Another observation is, the syntaxes offer differing levels of expressiveness (and need for expressiveness) between each other; these differences manifest in how the match-ups between child nodes must be handled. We see that CS rules can be much harder to write than either AS rules or FS rules. Thus, we begin to see that writing three separate rule sets for every new language is a great burden for the user.

One may have noticed from the above example that the FS rules are very similar to the CS rules. In fact, one might wonder if the CS rules are redundant—if Schütz itself could infer

the CS rule set from the FS rule set, thus saving the language definer a significant amount of painstaking clerical work. It turns out this is, indeed, the case—the CS rule set can be inferred from the FS rule set (with the help of the AS rule set). This mechanism does not currently exist, however; we must create it.

In this paper we present a method for mechanically generating a set of complete and correct CS rules from sets of complete and correct FS rules and AS rules. The remainder of this paper is organized in the following way: In chapter 2 we discuss LDL in greater detail and present our method for mechanically generating CS rule sets. In chapter 3 we prove this method to be correct. In chapter 4 we give some concluding remarks.

CHAPTER 2

METHOD FOR GENERATING CS RULE SET

In this chapter we discuss LDL in greater detail, then discuss a method for mechanically generating a CS rule set from an FS rule set and AS rule set. We use the following conventions: FS rules and their LDL definitions will appear in `monospace`. General pseudocode will appear in *italics*, while pseudocode that generates sentence fragments (discussed later) or CS rules will appear in *`italicized monospace`*.

2.1 Details of LDL

The CS, AS, and FS are used to define sets of fixed rules and list rules, each rule with its own complement of principal children. Fixed and list rules can be thought of as root nodes with complements of child nodes, and as such, the terms "child rule" and "child node" will be used interchangeably in this paper. AS and FS principal children may have a type. In fixed rules, the types of the children may be different, but the number and order of children is fixed. In list rules, the types of the children must all be the same, but the number and order of children may vary. There are three flavors of list rules: star-list rules, which may have zero or more children; plus-list rules, which may have one or more children; and plural-list rules, which may have two or more children. Not all flavors are available in all syntaxes. In programming language parlance, fixed rules are like C structs and Pascal records, while list rules are like arrays. For the CS, AS, and FS, respectively, LDL's names for fixed rules are `CsFixedRule`, `AsFixedRule`, and `FsFixedRule`; for list rules they are `CsListRule`, `AsListRule`, and, `FsListRule`; and for principal child rules they are `CsChild`, `AsChild`, and `FsChild`.

The AS rules of a particular language definition determine which CS rules and FS rules must also appear in that language definition. The CS rule set is expected to provide at least one

rule for building each AS rule. A `CsFixedRule` may build an `AsFixedRule`. Less obviously, a `CsFixedRule` may also build an `AsListRule`, and a `CsListRule` may build an `AsListRule` of equal or lesser minimum cardinality. For example, a `CsPlusRule`-style `CsListRule` may build an `AsStarRule`- or `AsPlusRule`-style `AsListRule`. The FS rule set is more heavily constrained. Each `FsFixedRule` must correspond to exactly one `AsFixedRule`, and each `FsListRule` must correspond to exactly one `AsListRule` (although an FS rule need not [and cannot] distinguish between star rules and plus rules). An FS rule and AS rule correspond when their LHSs are the same and their child types match. Their child types match when they appear in the same order in both the FS rule and AS rule (ignoring FS children that contribute solely to formatting). If an FS rule includes child names, those too must match the corresponding AS rule's child names.

LDL is defined in itself; that is, there's a language definition in LDL of LDL. This means there are CS, AS, and FS rule sets for LDL itself. In this paper, we deal heavily with the CS rule set in that language definition, particularly the subset of CS rules defining the FS. We shall call elements of that subset *FS components* to distinguish them from the CS rules or FS rules of an arbitrary language definition (i.e., one that may not be for LDL itself).

2.2 An intuitive method for mechanically generating CS rule sets

Intuitively, given an AS rule and corresponding FS rule, generating a CS rule from them requires four steps. We'll illustrate these steps by revisiting our example language definition from chapter 1. We'll generate CS rules for the `Times` node type. For clarity, we'll simplify the accompanying AS rule:

```
Times :: Left : Expr ; Right : Expr .
Times -> MEMBER Plus ( "(" _ ")" ) "*"
         MEMBER Plus ( "(" _ ")" ) .
```

First, make a copy of the FS rule, change its *produces* meta-symbol to the CS analog, and remove all child names. This forms the base of our generated CS rule:

```
Times ::= MEMBER Plus ( "(" _ ")" ) "*"
          MEMBER Plus ( "(" _ ")" ) .
```

Second, strip all the pure formatting tokens out of the copy. Examples of such tokens are `!`, signifying a linebreak; and `@ 2`, signifying an indention level of two. As there are none of these tokens in our example, the copy does not change:

```
Times ::= MEMBER Plus ( "(" _ ")" ) "*"
          MEMBER Plus ( "(" _ ")" ) .
```

Third, expand conditionals into multiple CS rules. In our example, we'll use a place-holder alternation, `NotPlus`. `NotPlus` holds the place for any child node type allowed by the corresponding AS rule, except `Plus`. Each of an FS rule's child's type may be a superset of those of the corresponding AS rule's corresponding child's type, but we need to know the actual type of the AS rule's corresponding child to create a valid CS rule—else, the parser might try to build TST nodes which have unsupported child types. We cannot build CS rules using only AS rules, however, because AS rules contain no information about insertion tokens; thus, we need both AS rules and FS rules. Note that the resulting generated rule set is nearly identical to the hand-written one we presented in chapter 1:

```
NotPlus ::= Times | Id .
Times ::= "(" Plus ")" "*" "(" Plus ")" .
Times ::= "(" Plus ")" "*" NotPlus .
Times ::= NotPlus "*" "(" Plus ")" .
Times ::= NotPlus "*" NotPlus .
```

Fourth, flatten all classes (and place-holder alternations), further expanding the CS rule set as needed. As a consequence of this, we can remove any place-holder alternations we made. Here's the final generated CS rule set for Times:

```
Times ::= "(" Plus ")" "*" "(" Plus ")" .
Times ::= "(" Plus ")" "*" Times .
Times ::= "(" Plus ")" "*" Id .
Times ::= Times "*" "(" Plus ")" .
Times ::= Id "*" "(" Plus ")" .
Times ::= Times "*" Times .
Times ::= Times "*" Id .
Times ::= Id "*" Times .
Times ::= Id "*" Id .
```

We choose to flatten classes and place-holder alternations for consistency. In the event the LHS of an FS rule is a class, we need to flatten it no matter what because the LHS of a CS rule cannot be a class. Thus, we choose to always flatten, rather than flatten only under certain conditions. As an added benefit, we reduce the amount of post-processing required to prepare the generated CS rule set for consumption by a parser generator. The resultant parser may also be faster, but larger; such concerns are outside the scope of this paper, however.

To generate a complete CS rule set, we'd apply this method to each FS rule (and associated AS rule) in the language definition.

2.3 A practical method for mechanically generating CS rule sets

The intuitive method for mechanically generating CS rule sets is appealing to humans because it's easy to understand, but is unsuitable for a computer due to the fuzziness of the steps involved. Thus, we present a different, but functionally equivalent method in this section, suitable for use by a computer.

We assume in describing this method that an arbitrary set of AS rules and a corresponding set of FS rules have been parsed and are available for reference. We assume the language definition is parsed as it would normally be, except that all handwritten CS rules are discarded and all `AsCsClassRules` are interpreted as `AsClassRules`. (The generated CS rule set will contain no class definitions.)

We present the *CS rule set generator* module as our method for mechanically generating CS rule sets. The CS rule set generator takes as input an AS rule set and FS rule set, and emits as output a CS rule set. For each FS rule in the input FS rule set, the component algorithm (described later) for `FsRule` is run; for generality, we refer to the `FsRule` algorithm simply as "the parent algorithm," since the same procedure is used for all component algorithms. Generally, the parent algorithm first calls in order of leftmost child to rightmost child the component algorithms associated with each of its children; note that these are component children (i.e., children in the FS rule component), not necessarily principal children. These calls may generate CS rules, and will lead to the return of CS rule fragment sets (parts of CS rules, as described later). Then, the parent algorithm assembles those fragment sets into another CS rule fragment set, which the parent algorithm returns to its own caller. All CS rules generated are added to a global set of CS rules. The output of the CS rule set generator after all FS rules have been processed is that global CS rule set.

A *component algorithm* is a subroutine associated with an FS component. These subroutines may generate whole CS rules, which are added to a global set. Each component algorithm returns a set of *CS rule fragments*. A *CS rule fragment* is part of a CS rule—a concatenation of terminals and non-terminals of the defined language. Parent component algorithms may concatenate elements of their children's fragment sets together into new fragment sets or whole CS rules.

Sometimes a component algorithm will return a fragment set containing the empty string, ε; typically when this happens, ε is that set's only member. Concatenation with the empty string works as expected: $\varepsilon A = A\varepsilon = A$ for any string A. The decision to return $\{\varepsilon\}$ instead of $\varnothing$ is typically made of convenience—it's more readable to express concatenation

with the former than the latter, and it makes our proofs of correctness in chapter 3 simpler and more readable.

In our component algorithms, any non-terminal prefixed with `Mg`, such as `MgChild` or `MgAltRule`, should be considered a place-holder for a unique non-terminal that is mechanically generated each time the component algorithm is executed. The mechanically generated non-terminal should be different from all other handwritten and mechanically generated non-terminals in the language definition and generated CS rule set. For instance, say we have the following function:

F := ∅
add to set F the fragments:
  `MgIdent`
generate CS rules:
  `MgIdent ::= .`
return F

In practice, `MgIdent` would be replaced with some unique, implementation-defined non-terminal, such as `__NT1337__`. So, the actual fragment generated would be `__NT1337__`, and the actual CS rule generated would be `__NT1337__ ::= .`. The next time the function is called, `MgIdent` would stand for a different unique non-terminal—say, `__NT1338__`.

In many component algorithms (and a couple of proofs in chapter 3), we need to *flatten* sets and classes. Note that classes can be represented as sets of other classes, `Idents`, and `Strings`. If S is a set or class represented as a set, and s_i is the i^{th} element of S, then flatten(S) = flatten(s_1) ∪ flatten(s_2) ∪ ⋯ ∪ flatten(s_n). If X is neither a set nor a class, then flatten(X) = { X }.

In the following sections of this chapter, we present the component algorithms, grouped by related FS components. In each section we first list the FS components, then show some example usages of those components, and then give pseudocode for the component algorithms. All examples come from M3.ldl1, written by Dr. Rodney Bates.

To aid the reader in understanding the FS components, we present a brief refresher on LDL's syntax: Each of the three primary syntaxes has at least one fixed rule and at least one list rule. In the CS, fixed rules are defines using the `::=` symbol, in the AS by using `::`, and the in FS by using `->`. List rules come in three varieties: star rules, in which the LHS derives zero or more of the right-hand side (RHS); plus rules, in which the LHS derives one or more of the RHS; and plural rules, in which the LHS derives two or more of the RHS. The CS uses all three, via the symbols `::=*`, `::=+`, and `::=++`, respectively. The AS uses only the first two, via the symbols `::*` and `::+`, respectively. The FS uses only the first, via the symbol `->*`. All fixed rules and list rules end with a period.

CS fixed rules may use alternation via the `|` symbol, or concatenation via the lack of separating symbols. This means a fixed rule may use either alternation or concatenation, but not both. If alternation is used, each terminal and non-terminal must be separated by a `|`. If concatenation is used, optional subsequences may be embedded in non-nesting square brackets `[` and `]`.

CS list rules always have a sequence of at least one terminal or non-terminal in their RHS. This sequence may be followed by a `||` symbol and a sequence of terminals which are to be used as list separators. For instance, `A || "/"` indicates a list of of `A`s, with each `A` separated from the other `A`s in the list by a `/` symbol.

We now present the component algorithms.

2.3.1 FsRule

```
FsRule ::= FsFixedRule | FsListRule .
```

`FsRule` is the top-level FS rule. There may be many `FsRules` in a language definition. See the Format Fixed Rules section and Format List Rules section for explanations and examples of `FsFixedRule` and `FsListRule`, respectively.

`FsRule`'s algorithm generates no CS rules and returns only its RHS's fragment set (which should be empty).

2.3.2 Format Fixed Rules

```
FsFixedRule
  ::= FsFixedDefaultRule | FsFixedHorizRule | FsFixedVertRule
    | FsFixedFillRule
FsFixedDefaultRule
  ::= IdentPlusList "->" FsFixedChildList "." .
FsFixedHorizRule
  ::= IdentPlusList "HORIZ" "->" FsFixedChildList "." .
FsFixedVertRule
  ::= IdentPlusList "VERT" "->" FsFixedChildList "." .
FsFixedFillRule
  ::= IdentPlusList "FILL" "->" FsFixedChildList "." .
```

`FsFixedRule`'s four alternatives, collectively called "format fixed rules," each define one or more `Idents` (`IdentPlusList`) as an FS node type having a complement of children fixed in class, position, and number. Each `Ident` must be defined as an abstract fixed rule elsewhere in the same language definition, and the children in the format fixed rule must match up to those in the associated abstract fixed rule as discussed earlier in this chapter. The format fixed rule's children are formatted according to a format kind, which may be explicitly stated via the `HORIZ`, `VERT`, or `FILL` keywords. Format kinds are not used by the CS, so our algorithm ignores them.

Here's an example `FsFixedVertRule`:

```
AsInterface VERT ->
     Safety : AsUnsafe
          RwINTERFACE
     Name : Id
     !
     ":"
     Imports : AsImportList
     Decls : AsDeclList
          RwEND
     FinalName : Id
     !
     "."
.
```

This rule defines `AsInterface` as a format fixed rule that formats its children using the `VERT` format kind. `Safety : AsUnsafe` is the first child node, `RwINTERFACE` the second, and `Name : Id` the third. The next symbol, `!`, stands for a particular kind of whitespace (a line break), and will be ignored by our algorithm. The `":"` symbol is not part of the AST or, thus, the associated abstract fixed rule, but must be included in the CS rules we generate. `Imports : AsImportList` is the fourth child, `Decls : AsDeclList` the fifth, `RwEND` the sixth, and `FinalName : Id` the seventh and last. The second `!` is a layout instruction for the formatter. The `"."` symbol is an insertion token.

`FsFixedRule`'s algorithm generates no CS rules and returns only its RHS's fragment set.

All four format fixed rules share this same algorithm:

for each fragment F_1 from IdentPlusList:
  for each fragment F_2 from FsFixedChildList:
    generate CS rule:
      F_1 `::=` F_2 `.`
return $\emptyset$

2.3.3 Format List Rules

```
FsListRule
  ::= FsListDefaultRule | FsListHorizRule | FsListVertRule
    | FsListFillRule .
FsListDefaultRule
  ::= IdentPlusList "->*"
      FsChild [ "||" FormatterList ] "." .
FsListHorizRule
  ::= IdentPlusList "HORIZ" "->*"
      FsChild [ "||" FormatterList ] "." .
FsListVertRule
  ::= IdentPlusList "VERT" "->*"
      FsChild [ "||" FormatterList ] "." .
FsListFillRule
  ::= IdentPlusList "FILL" "->*"
      FsChild [ "||" FormatterList ] "." .
```

`FsListRule`'s four alternatives, collectively called "format list rules," each define one or more `Idents` (`IdentPlusList`) as an FS node type having a variable number of children of one class, denoted by the `FsChild`. Each `Ident` must be defined as an abstract list rule elsewhere in the same language definition, and the `FsChild` must refer to the same node types referred to by the abstract list rule's non-optional `Ident` or `IdentPlusList`, as suggested earlier in this chapter. The format list rule's children are formatted according to a format kind, which may be explicitly stated via the `HORIZ`, `VERT`, or `FILL` keywords. Format kinds are not used by the CS, so our algorithm ignores them. The optional `FormatterList` indicates symbols that should separate items in the list.

Here's an example `FsListFillRule`:

```
AsASImportList FILL ->*
     Elems : INDENT @ 2 AsCsClASImportItem || ! "," .
```

This rule defines `AsASImportList` as a format list rule that formats its children using the `FILL` format kind. `Elems : INDENT @ 2 AsCsClASImportItem` is the `FsChild`,

defining `AsASImportList` to be a list of children of class `AsCsClASImportItem`.

`INDENT @ 2` and `!` are layout instructions to the formatter. The "`,`" symbol represents an insertion token.

`FsListRule`'s algorithm generates no CS rules and returns only its RHS's fragment set.

All four format list rules share this same algorithm:

for each fragment F_2 from FsChild:
  generate CS rule:
    `MgIdent ::= F2 .`
for each fragment F_1 from IdentPlusList:
  with the fragment F_3 from FormatterList:
    if $F_3 = \{\varepsilon\}$:
      if the corresponding AsListRule is ::-type:*
        generate CS rule:
          `F1 ::=* MgIdent .`
      else if the corresponding AsListRule is ::+-type:
        generate CS rule:
          `F1 ::=+ MgIdent .`
    else if $F_3 \neq \{\varepsilon\}$:
      if the corresponding AsListRule is ::-type:*
        generate CS rule:
          `F1 ::=* MgIdent || F3 .`
      else if the corresponding AsListRule is ::+-type:
        generate CS rule:
          `F1 ::=+ MgIdent || F3 .`
return $\emptyset$

2.3.4 FsFixedChildList

```
FsFixedChildList ::=* FsFixedChild .
FsFixedChild ::= FsChild | LineBreak | Indenter | String .
```

An `FsFixedChildList` is a concatenation of principal children (`FsChild`), insertion tokens (`String`), and/or formatting information (`LineBreak`, `Indenter`). See the Format Fixed Rules section for an example of both of these rules in action.

`FsFixedChild`'s algorithm generates no CS rules and returns only its RHS's fragment set.

`FsFixedChildList`'s algorithm is as follows: Let n be the number of `FsFixedChild`ren in the RHS:

$F_{FsFixedChildList} := \emptyset$
for each fragment F_1 *from* $FsFixedChild_1$*:*
  for each fragment F_2 *from* $FsFixedChild_2$*:*
    ...
      for each fragment F_n *from* $FsFixedChild_n$*:*
        add to $F_{FsFixedChildList}$ *the fragment:*
          $F_1 \ F_2 \ \ldots \ F_n$
if $F_{FsFixedChildList} = \emptyset$ *:*
  return $\{\varepsilon\}$
else if $F_{FsFixedChildList} \neq \emptyset$ *:*
  return $F_{FsFixedChildList}$

2.3.5 FsChild

```
FsChild ::= FsChildPlain | FsSubtree | FsCondFmt | FsChildCase .
```

An `FsChild` is generally the top-level FS component for a format principal child. There are exceptions, such as the special case of `FsChildPlain` place-holding a `String`, which are discussed in the sections specific to each of `FsChild`'s four alternatives.

Examples of `FsChild` in action are given in most other examples used in this chapter.

`FsChild`'s algorithm generates no CS rules and returns only its RHS's fragment set.

2.3.6 FsChildPlain

```
FsChildPlain
  ::= [ Ident ":" ] [ "INDENT" Indenter ] IdentOrDontCare .
FsChildPlain
  ::= [ Ident ":" ] [ "INDENT" Indenter ] "{" Alternation "}" .
IdentOrDontCare ::= Ident | DontCare .
DontCare ::= '_' .
```

An `FsChildPlain` normally defines a format principal child. It should match up to an abstract principal child, as discussed earlier in this chapter. The first `Ident`, which is optional, is the child's name. If given, it must match the associated abstract principle child's name, but our

algorithm does not use it. The optional "INDENT" Indenter deals with whitespace. The IdentOrDontCare or Alternation indicates the set of allowed node types for this format principal child, and must be a superset of the allowed node types indicated by the associated abstract principal child. The DontCare alternative, in particular, functions as a wildcard for node types.

There is a special case of FsChildPlain where the two optional parts are absent, and either IdentOrDontCare derives an Ident aliasing just a String, or Alternation derives just such an Ident or just a String. In either case, the FsChildPlain does not represent a format principal child, and thus, does not match up to an associated abstract principal child. Instead, the FsChildPlain functions as an insertion token.

FsChildPlain's IdentOrDontCare and Alternation allowing a superset of the types allowed by the corresponding abstract principal child causes no problems when translating an AST node back to edited text, since in that case the superfluous types are never used. CS rules allowing those superfluous types, however, would be too permissive, accepting languages unsupported by the AST. As such, our algorithm uses the corresponding abstract principal child's allowed node types, rather than the FsChildPlain's.

In practice, an abstract principle child is defined via AsChild, AsReqdChild, and AsOptChild, which are defined by the following CS rules:

```
AsChild ::= AsReqdChild | AsOptChild .
AsReqdChild ::= [ Ident ":" ] Ident .
AsReqdChild ::= [ Ident ":" ] "{" Alternation "}" .
AsOptChild ::= [ Ident ":" ] "[" Ident "]" .
AsOptChild ::= [ Ident ":" ] "[" "{" Alternation "}" "]" .
```

As in FsChildPlain, the optional Ident is the child's name, and the following Ident or Alternation indicates the child's allowed node types. We refer to both

AsReqdChild and AsOptChild as AsChild in our algorithm to emphasize the one-to-one correspondence between an FsChildPlain and its associated abstract principal child.

Both the Format Fixed Rules and Format List Rules sections contain examples of the IdentOrDontCare case of FsChildPlain in action. We can rework the Format List Rules example into an Alternation case without changing the rule's meaning like so:

```
AsASImportList ->* FILL
     Elems : INDENT @ 2 { Id | AsASItem } || ! ",” .
```

IdentOrDontCare's algorithm generates no CS rules and returns only its RHS's fragment set.

DontCare's algorithm is as follows:

return $\{\varepsilon\}$

FsChildPlain's algorithm uses fragment sets from the corresponding AsChild, rather than FsChildPlain itself. This means we run the AsChild's Ident or IdentPlusList through our respective algorithm. Here's FsChildPlain's algorithm:

if special case (see above):
return IdentOrDontCare's or Alternation's fragment set
else:
discard IdentOrDontCare's or Alternation's fragment set
$F_{FsChildPlain}$:= Ø
if the corresponding AsChild names types with Ident:
for each fragment F from AsChild's Ident:
add to $F_{FsChildPlain}$ the fragment:
F
else if the corresponding AsChild names types with [Ident]:
add to $F_{FsChildPlain}$ the fragments:
$MgIdent_0$
$MgIdent_1$
generate CS rule:
$MgIdent_0$ ::= .
for each fragment F from AsChild's Ident:
generate CS rule:
$MgIdent_1$ ::= F .
else if the corresponding AsChild names types with { Alternation }:
for each fragment F from AsChild's Alternation:
add to $F_{FsChildPlain}$ the fragment:
F
else if the corresponding AsChild names types with [{ Alternation }]:
add to $F_{FsChildPlain}$ the fragments:
$MgIdent_0$
$MgIdent_1$
generate CS rule:
$MgIdent_0$ ::= .
for each fragment F from AsChild's Alternation:
generate CS rule:
$MgIdent_1$ ::= F .
return $F_{FsChildPlain}$

2.3.7 Format Subtree Rules

```
FsSubtree
  = FsDefaultSubtree
  | FsHorizSubtree | FsVertSubtree | FsFillSubtree .
FsDefaultSubtree ::= "(" FsFixedChildList ")" .
FsHorizSubtree ::= "HORIZ" "(" FsFixedChildList ")" .
FsHorizSubtree ::= "HORIZ" FsCondFmt .
FsVertSubtree ::= "VERT" "(" FsFixedChildList ")" .
FsVertSubtree ::= "VERT" FsCondFmt .
FsFillSubtree ::= "FILL" "(" FsFixedChildList ")" .
FsFillSubtree ::= "FILL" FsCondFmt .
```

Each of FsSubtree's four alternatives can be used to specify a subsequence of children as having a different format kind (HORIZ, VERT, FILL) from their parent. Format kinds influence the formatter's emission of whitespace.

FsSubtree's algorithm generates no CS rules and returns only its RHS's fragment set.

Each of FsSubtree's four alternatives' algorithms generates no CS rules and returns only its principal child's fragment set.

2.3.8 Format Conditional Rules

```
FsChildCase
  ::= "CASE" FsChildPlain "OF" FsCondFmtList
      [ "ELSE" FsFixedChildList ] "END" .
FsCondFmtList ::=+ FsCondFmt || "|" .
FsCondFmt
  ::= FsCondAbsent | FsCondPresent | FsCondEmpty
    | FsCondNonempty | FsCondPlural | FsCondNonplural
    | FsCondMember .
FsCondAbsent ::= "ABSENT" "(" FsFixedChildList ")" .
FsCondPresent ::= "PRESENT" "(" FsFixedChildList ")" .
FsCondEmpty ::= "EMPTY" "(" FsFixedChildList ")" .
FsCondNonempty ::= "NONEMPTY" "(" FsFixedChildList ")" .
FsCondPlural ::= "PLURAL" "(" FsFixedChildList ")" .
FsCondNonplural ::= "NONPLURAL" "(" FsFixedChildList ")" .
FsCondMember ::= "MEMBER" Ident "(" FsFixedChildList ")" .
FsCondMember
  ::= "MEMBER" "{" Alternation "}" "(" FsFixedChildList ")" .
```

A format conditional rule uses one or more predicates (ABSENT, PRESENT, etc.) to define multiple ways of formatting an abstract principal child.

In an FsChildCase, the FsChildPlain following the CASE keyword indicates the abstract principal child to be formatted. The formatter tests the predicates listed in FsCondFmtList in the order the language designer wrote them, and the abstract node is formatted according to the first rule satisfied. If no rule is satisfied, the abstract node is

formatted according to ELSE's FsFixedChildList. The ELSE clause may be omitted if FsCondFmtList covers all possible children (thus leaving the ELSE clause unreachable).

The seven predicates can be divided into two groups. ABSENT, PRESENT, and MEMBER apply only to fixed-rule abstract principal children. EMPTY, NONEMPTY, PLURAL, and NONPLURAL apply only to list-rule abstract principal children.

ABSENT is true if the child is missing from the AST; PRESENT is true if the child exists. MEMBER is true if the child's type is in the set of node types indicated by the Ident or Alternation immediately following the MEMBER keyword.

EMPTY is true if the child contains exactly zero nodes; NONEMPTY is true if the child contains one or more nodes. Likewise, PLURAL is true if the child contains two or more nodes; NONPLURAL is true if the child contains exactly zero or one nodes.

Following each predicate keyword is an FsFixedChildList. This FsFixedChildList must contain exactly one format principal child, and that child must match the abstract principal child to be formatted. Beyond this requirement, the FsFixedChildList may contain any number of insertion tokens and pure formatting components. In the discussions and algorithms that follow, it will be helpful to split these FsFixedChildLists into three parts: α, the FsFixedChildren before the format principal child; B, the format principal child itself; and γ, the FsFixedChildren after the format principal child.

In our algorithm for FsChildCase, we expand the given ELSE clause, if any, into an extension to the list of cases. Say we have this general form of ELSE clause:

ELSE ($\text{FsFixedChildList}_{\text{ELSE}}$)

If the format principal child is a fixed rule, the ELSE clause is expanded to:

```
| ABSENT ( FsFixedChildList_ELSE )
| PRESENT ( FsFixedChildList_ELSE )
```

If the format principal child is a list rule, the ELSE clause is expanded to:

```
| EMPTY ( FsFixedChildList_ELSE )
| NONEMPTY ( FsFixedChildList_ELSE )
```

As syntactic sugar, Schütz allows `FsCondFmt`s to be used outside of `FsCondFmtList`s. Such `FsCondFmt`s are said to be "solitary." Let `COND` stand for any of `ABSENT`, `PRESENT`, `EMPTY`, `NONEMPTY`, `PLURAL`, `NONPLURAL`, `MEMBER Ident`, or `MEMBER { Alternation }`. Then, a solitary `FsCondFmt`,

```
COND ( α B γ )
```

is equivalent to,

```
CASE B OF COND ( α B γ ) ELSE ( B ) END
```

Our algorithms treat solitary `FsCondFmt`s as such; and in fact, this is how Dr. Rodney Bates' implementation of the formatter handles solitary `FsCondFmt`s.

Here's an example of a solitary `FsCondPresent`, embedded in a format fixed rule:

```
AsConstDecl
     VERT -> ( Name : Id
          PRESENT ( ! @ 4 ":" Type : INDENT @ 6 AsType )
          ( ! @ 4 "=" Value : INDENT @ 6 AsExpr )
     )
     !
     ";"
.
```

We're interested in the third line. This line uses an `FsCondPresent` to define formatting for `Type : AsType`, principal child, which is the second principal child of `AsConstDecl`. If `AsType` is present, the formatter will emit some whitespcae, designated by `! @ 4` and `IDENT`

`@ 6`, along with the string literal `":"` and whatever `AsType` formats to. If `AsType` is absent, the formatter will emit nothing at all.

Let's change the third line to a solitary `FsCondAbsent`:

```
ABSENT ( ! @ 4 ":" Type : INDENT @ 6 AsType )
```

In this case, if `AsType` is present, the formatter will emit only what `AsType` formats to; if `AsType` is absent, the formatter will emit the formatting and string literal. Whatever `AsType` formats to would be emitted, too, were it not absent!

Here's an example `FsChildCase` using the `MEMBER` predicate, embedded in a format list rule:

```
AsImportList VERT ->*
     CASE Elems : AsImport
     OF MEMBER AsASImportOrList
          ( ( RwIMPORT @ 4 Elems : INDENT @ 2 AsImport ) ! ";" )
     ELSE ( Elems : INDENT @ 2 AsImport ) ! ";"
     END
.
```

This `FsChildCase` formats an `AsImport` principal child named `Elems`. If that child is a member of the class `AsASImportOrList`, then it will be formatted with `( RwIMPORT @ 4 Elems : INDENT @ 2 AsImport ) ! ";"`. `RwIMPORT` is an `Ident` aliasing a `String`, and `";"` is a string literal. The symbols `@ 4`, `IDENT @ 2`, and `!` are layout information. If the `AsImport` node is not a member of `AsASImportOrList`, it will be formatted with `( Elems : IDENT @ 2 AsImport ) ! ";"`.

The above example will be expanded for processing by our algorithm to the following:

```
AsImportList VERT ->*
     CASE Elems : AsImport
     OF MEMBER AsASImportOrList
          ( ( RwIMPORT @ 4 Elems : INDENT @ 2 AsImport ) ! ";" )
     | ABSENT ( ( Elems : INDENT @ 2 AsImport ) ! ";" )
     | PRESENT ( ( Elems : INDENT @ 2 AsImport ) ! ";" )
     END
.
```

Blindly generating CS rules from the `FsCondFmtList` would lead to ambiguities in the CS rule set since, in some cases, it's possible a child satisfies two or more format conditional rules. This would lead to two or more CS rules that would reduce to the same abstract node. For instance, were the order of predicates in the `FsChildCase` `PLURAL` and `NONEMPTY`, both of the resultant CS rules would accept children with two or more nodes, instead of just the `PLURAL` predicate's generated CS rule.

To prevent ambiguity in the generated CS rule set, `FsChildCase`'s algorithm maintains a state to track which possibilities have been covered. For instance, in the above example, after generating the CS rule for the `PLURAL` case, `FsChildCase`'s state would be updated to reflect that it'd covered the case of an abstract child with two or more nodes. The algorithm for the `NONEMPTY` case would then know to generate a CS rule only for children with exactly one node, instead of children with one or more nodes.

`FsChildCase`'s state uses two sets: a quantifier set and a node set. A quantifier set Q tracks which quantities of the abstract node have not participated in a CS rule generation. $Q \subseteq \{ 0, 1, 2+ \}$. A node set C tracks which of the allowed abstract node types have not participated in a CS rule generation. $C \subseteq$ flatten({ x | x is an abstract node type }).

`FsCondFmtList`'s algorithm generates no CS rules and returns the union of its children's fragment sets.

`FsCondFmt`'s algorithm generates no CS rules and returns only its RHS's fragment set.

In the following algorithms, if L is a list rule, then let $L_{FsChild}$ stand for the fragment set returned for L's `FsChild`, and $L_{FormatterList}$ stand for the fragment set returned for L's `FormatterList`. If L has no `FormatterList`, let $L_{FormatterList} = \{\varepsilon\}$. Note that, even if L has a `FormatterList`, it's possible that $L_{FormatterList} = \{\varepsilon\}$.

`FsChildCase`'s algorithm is lengthy, so will be annotated with comments. Comments begin with a (* and end with a *).

(Initialize our set of fragments to be returned *)*
$F_{FsChildCase} := \varnothing$

(Initialize our state *)*
if FsChildPlain names a format fixed rule:
  Q := { 0 }
  C := flatten(the corresponding AsChild's child name Ident)
else if FsChildPlain names a format list rule:
  if the corresponding abstract principal child is a star list:
    Q := { 0, 1, 2+ }
  else if the corresponding abstract principal child is a plus list:
    Q := { 1, 2+ }
  C := $\varnothing$

(Expand the ELSE clause, if any *)*
if there is an ELSE clause:
  if FsChildPlain names a format fixed rule:
    append to FsCondFmtList the two predicates:
      ABSENT (FsFixedChildList$_{ELSE}$ *)*
      PRESENT (FsFixedChildList$_{ELSE}$ *)*
  else if FsChildPlain names a format list rule:
    append to FsCondFmtList the two predicates:
      EMPTY (FsFixedChildList$_{ELSE}$ *)*
      NONEMPTY (FsFixedChildList$_{ELSE}$ *)*

(Process our FsCondFmtList *)*
for each fragment set F from FsCondFmtList (in the order they appear in the instance):
  add to $F_{FsChildCase}$ *all elements of the fragment set:*
    F

(Done *)*
return $F_{FsChildCase}$

Each predicate needs a different algorithm. However, every predicate's algorithm should be wrapped in the following code:

$F_{FsCondFmtList} := \emptyset$
(predicate-specific code here... *)*
return $F_{FsCondFmtList}$

Here are the algorithms:

2.3.8.1 FsCondAbsent

if $Q \ni 0$:
  $F_{FsCondFmtList} := \{ \alpha$ `MgChild` $\gamma \}$
  generate CS rule:
    `MgChild ::= .`
  $Q := Q - \{ 0 \}$

2.3.8.2 FsCondPresent

if $C \neq \emptyset$:
  $F_{FsCondFmtList} := \{ \alpha$ `MgChild` $\gamma \}$
  for each element c of C:
    generate CS rule:
      `MgChild ::= c .`
  $C := \emptyset$

2.3.8.3 FsCondMember

A := the fragment set returned for our Ident or Alternation, whichever was used
$T := A \cap C$
if $T \neq \emptyset$:
  $F_{FsCondFmtList} := \{ \alpha$ `MgChild` $\gamma \}$
  for each element t of T:
    generate CS rule:
      `MgChild ::= t .`
  $C := C - T$

2.3.8.4 FsCondEmpty

if $Q \ni 0$:
  $F_{FsCondFmtList} := \{ \alpha$ `MgChild` $\gamma \}$
  for each element b of flatten(B):
    generate CS rule:
      `MgChild BUILD b ::= .`
  $Q := Q - \{ 0 \}$

2.3.8.5 FsCondNonempty

if $Q \ni 1$*:*
  $F_{FsCondFmtList} := \{ \alpha$ `MgChild` $\gamma \}$
  for each element b of flatten(B):
    generate CS rule:
      `MgChild BUILD b ::=` $b_{FsChild}$ `.`
  $Q := Q - \{ 1 \}$
if $Q \ni 2+$*:*
  $F_{FsCondFmtList} := \{ \alpha$ `MgChild` $\gamma \}$
  if $FormatterList_B = \{\varepsilon\}$ *:*
    for each element b of flatten(B):
      generate CS rule:
        `MgChild BUILD b ::=++` $b_{FsChild}$ `.`
  else if $FormatterList_B \neq \{\varepsilon\}$ *:*
    for each element b of flatten(B):
      generate CS rule:
        `MgChild BUILD b ::=++` $b_{FsChild}$ `||` $b_{FormatterList}$ `.`
  $Q := Q - \{ 2+ \}$

2.3.8.6 FsCondPlural

if $Q \ni 2+$*:*
  $F_{FsCondFmtList} := \{ \alpha$ `MgChild` $\gamma \}$
  if $FormatterList_B = \{\varepsilon\}$ *:*
    for each element b of flatten(B):
      generate CS rule:
        `MgChild BUILD b ::=++` $b_{FsChild}$ `.`
  else if $FormatterList_B \neq \{\varepsilon\}$ *:*
    for each element b of flatten(B):
      generate CS rule:
        `MgChild BUILD b ::=++` $b_{FsChild}$ `||` $b_{FormatterList}$ `.`
  $Q := Q - \{ 2+ \}$

2.3.8.7 FsCondNonplural

if $Q \ni 0$*:*
  $F_{FsCondFmtList} := \{ \alpha$ `MgChild` $\gamma \}$
  for each element b of flatten(B):
    generate CS rule:
      `MgChild BUILD b ::= .`
  $Q := Q - \{ 0 \}$
if $Q \ni 1$*:*
  $F_{FsCondFmtList} := \{ \alpha$ `MgChild` $\gamma \}$
  for each element b of flatten(B):
    generate CS rule:
      `MgChild BUILD b ::=` $b_{FsChild}$ `.`
  $Q := Q - \{ 1 \}$

2.3.9 FormatterList

```
FormatterList ::=* Formatter .
Formatter ::= LineBreak | Indenter | Ident | String .
```

When formatting a format list rule that uses a `FormatterList`, the formatter emits the insertion tokens and whitespace specified by that `FormatterList` between each element of the formatted list. Lists with zero or one elements are formatted with no `Formatter`s, even if a `FormatterList` is given.

See the Format List Rules section for an example of both of these rules in action.

`FormatterList`'s algorithm is as follows:

$F_{FormatterList} := \emptyset$
for each fragment F_1 *from* $Formatter_1$*:*
  for each fragment F_2 *from* $Formatter_2$*:*
    ...
      for each fragment F_n *from* $Formatter_n$*:*
        add to $F_{FormatterList}$ *the fragment:*
          $F_1 \ F_2 \ \ldots \ F_n$
if $F_{FormatterList} = \emptyset$ *:*
  return $\{\varepsilon\}$
return $F_{FormatterList}$

`Formatter`'s algorithm generates no CS rules and returns only its RHS's fragment set.

2.3.10 Format Layout Rules

```
Indenter
  ::= BolIndenter | ParentIndenter | PosIndenter
    | BolPlusIndenter | ParentPlusIndenter | PosPlusIndenter
    | BolMinusIndenter | ParentMinusIndenter | PosMinusIndenter
.
BolIndenter ::= "^" [ Integer ] .
ParentIndenter ::= "@" [ Integer ] .
PosIndenter ::= "$" [ Integer ] .
BolPlusIndenter ::= "^" "+" Integer .
ParentPlusIndenter ::= "@" "+" Integer .
PosPlusIndenter ::= "$" "+" Integer .
BolMinusIndenter ::= "^" "-" Integer .
ParentMinusIndenter ::= "@" "-" Integer .
PosMinusIndenter ::= "$" "-" Integer .
```

`Indenter`'s nine alternatives, collectively called "format layout rules," are used to suggest how much indentation (whitespace) the formatter should emit at that point in the formatting. Whether or not the formatter actually emits the line break depends on the enclosing format kind, among other things. The parser does not discriminate indentation from other forms of token separation.

Examples of some of these format layout rules in action are given in most other examples used in this chapter.

`Indenter`'s algorithm generates no CS rules and returns only its RHS's fragment set.

Each format layout rule's algorithm generates no CS rules and returns only its principal child's fragment set.

2.3.11 Miscellaneous Rules

```
Alternation ::=+ CsAtom || "|" .
CsAtom ::= Ident | String .
LineBreak ::= "!" .
IdentPlusList ::=+ Ident || "|" .
Ident VARTERM .
String VARTERM .
Integer VARTERM .
```

In the FS, `Alternations` are used only to define anonymous classes. As such, each of an `Alternation`'s `CsAtom`s can only ever stand for an `Ident`, never a `String`.

In the FS, `CsAtom` is used only by `Alternation`.

A `LineBreak` in a format rule suggests the formatter emit a line break at that point in the formatting. Whether or not the formatter actually emits the line break depends on the enclosing format kind. The parser does not discriminate line breaks from other forms of token separation.

In the FS, `IdentPlusList` is used only on the LHS of FS rules.

`Ident`, `String`, and `Integer` are variably spelled LDL terminals. An instance of an `Ident` is spelled with a letter or underscore followed by zero or more letters, digits, and underscores. An instance of a `String` is spelled with a sequence of characters enclosed by double quotes. An instance of an `Integer` is spelled with any sequence of digits. When we refer to `Ident`, `String`, and `Integer` in their respective algorithms, we refer to the actual spelling of that terminal, not the LDL keyword itself.

`Alternation`'s algorithm is as follows:

$F_{Alternation} := \emptyset$
for each fragment F_1 *from* $CsAtom_1$*:*
  for each fragment F_2 *from* $CsAtom_2$*:*
    ...
      for each fragment F_n *from* $CsAtom_n$*:*
        add to $F_{Alternation}$ *the fragment:*
          $F_1 \ F_2 \ \ldots \ F_n$
return $F_{Alternation}$

`CsAtom`'s algorithm generates no CS rules and returns only its RHS's fragment set.

`LineBreak`'s and `Integer`'s algorithms are as follows:

return $\{\varepsilon\}$

`IdentPlusList`'s algorithm is as follows:

$F_{IdentPlusList}$:= Ø
for each fragment F_1 from $Ident_1$:
  for each fragment F_2 from $Ident_2$:
    ...
      for each fragment F_n from $Ident_n$:
        add to $F_{IdentPlusList}$ the fragment:
          F_1 F_2 ... F_n
return $F_{IdentPlusList}$

`Ident`'s algorithm is as follows:

return flatten(`Ident`)

`String`'s algorithm is as follows:

return {`String`}

CHAPTER 3

PROOF OF CORRECTNESS

We begin this chapter with a discussion about Schütz's three modules—the formatter, the generator, and the parser. Our main goal in this chapter, however, is to prove four theorems about our method. Theorem 1 deals with the correctness of component algorithms. Theorem 2 deals with the ability of CS rule sets generated from individual FS rule/AS rule pairs to be used by the parser to rebuilt formatted text syntax subtrees. Theorem 3 extends theorem 2 to full TSTs. Theorem 4 deals with the preservation of unambiguity of the FS rule set and AS rule set in the generated CS rule set.

Before we begin, it will be helpful to distinguish between FS components and FS rule components. *FS components* are part of LDL, whereas FS rule components are part of an FS rule in a language definition. An FS component is one of the CS rules defining LDL's FS; in chapter 2, we gave a component algorithm for each FS component. An *FS rule component*, on the other hand, is a part of an FS rule corresponding to an FS component. For instance, in our example language definition in chapter 1, we gave the following FS rule:

```
Times -> MEMBER Plus ( "(" _ ")" ) "*"
         MEMBER Plus ( "(" _ ")" ) .
```

LDL has an FS component called `FsCondMember`. There are two FS rule components in the example FS rule for `FsCondMember`, both spelled the same:

```
MEMBER Plus ( "(" _ ")" )
```

We discussed the formatter and parser modules of Schütz in chapter 1, and the generator module in chapter 2. Here, we restate the inputs, outputs, and behavior of those modules. Further, we will find it helpful to define fictitious, component-level versions of these modules for

use in our lemmas, theorems, and proofs in this chapter.

In this chapter, we shall call Schütz's formatter module the *tree formatter*. The tree formatter takes as input a TST, a set of AS rules, and a set of FS rules; it gives as output a sequence of insertion tokens (string literals and variably-spelled terminals) and whitespace. To format a TST, the tree formatter is called with the root of that TST. For generality, we refer to the root node of the TST simply as "the TST node," since the same procedure is used for all TST nodes. The tree formatter looks up the FS rule that corresponds to the TST node by matching the TST node's name (the LHS of its associated AS rule) with the LHS of an FS rule. The FS rule will list a sequence of insertion tokens, whitespace (indention, line breaks, etc.), and/or principal children. For the purposes of this proof, we treat any contiguous string of whitespace characters as a single token separator, since that's what the parser does. The FS rule is processed left-to-right. When the tree formatter reads an insertion token from the FS rule, it emits that insertion token to Schütz's editing window. When the tree formatter reads a principal child from the FS rule, it recurses on the corresponding TST node's child node. Thus, the tree formatter could be thought of as traversing the TST in postorder, since parent nodes are not completely "visited" until all of their children have been visited.

We'll find it convenient in this chapter to use a fictitious form of formatter called the *component formatter*. The component formatter takes as input an AST node (with some indication as to the current principal child), and an FS rule component; it gives as output a sequence of insertion tokens, principal children, and whitespace. The component formatter is similar to the tree formatter, except it treats principal children the same as insertion tokens; that is, the component formatter outputs principal children as insertion tokens rather than formatting them. Further, the component formatter can format using an FS rule component, in effect

formatting just part of a TST node. An FS rule component, in this context, is the subsequence of the full FS rule's sequence of insertion tokens, whitespace, and principal children associated with the FS rule component. We assume there is some mechanism for identifying any abstract principal children associated with the FS rule component.

Although the component formatter treats principal children the same as insertion tokens, we shall continue to refer to principal children separately from insertion tokens when discussing the component formatter. The reasons for this are, 1) the two are treated separately by the tree formatter, and 2) the two will be treated separately in later proofs that build off theorems and lemmas the rely on the component formatter.

Our *CS rule generator* module takes as input an AS rule set and FS rule set, and emits as output a CS rule set. For each FS rule in the input FS rule set, the component algorithm for `FsRule` is run; for generality, we refer to the `FsRule` algorithm simply as "the parent algorithm," since the same procedure is used for all component algorithms. Generally, the parent algorithm first calls the appropriate algorithms for each of its children, in order of leftmost child to rightmost child; note that these are component children, not necessarily principal children. These calls may generate CS rules, and will lead to the return of fragment sets. Then, the parent algorithm assembles those fragment sets into another fragment set, which the parent algorithm returns to its own caller. Although, in practice, the CS rules are added to a global set, we'll find it convenient in our proofs to assume the CS rules are passed up the call chain alongside the fragment sets (but not in those sets!); doing this does not change the outcome. In the case of `FsRule`'s algorithm, the result will be a non-empty set of CS rules and an empty fragment set. The output of the generator after all FS rules have been processed is the union of the generated CS rule sets.

In the generator, each FS rule component has an associated algorithm (these were given in chapter 2), which we call a *component algorithm*. We'll find it convenient in our proofs to talk about executing the algorithms for arbitrary FS rule components, not just `FsRule`. In this case, the output of said execution will be a possibly empty set of CS rules and a possibly non-empty fragment set.

In this chapter, we shall call the general parser module the *edited-text parser*. The edited-text parser takes as input a set of CS rules, a set of AS rules, and a string to parse (the edited text); it gives as output a TST. In reality, the edited-text parser is created using an LALR parser generator from a CFG translation of the CS rule set, and with semantic actions drawn from the input CS rule set and the AS rule set. Those implementation details are immaterial to this chapter, though, so we shall abstract them as follows: Let the input CS rule set represent a context-free grammar. Alongside parsing, the edited-text parser builds a TST. It does this using a build stack. Whenever a CS rule is applied, a TST node may be created. If the CS rule has a `BUILD` keyword, the `Ident` associated with that keyword is looked up in the AS rule set; if it matches an `Ident` in the flattened LHS of any AS rule, that AS rule is used as the template for the node (there should be one and only one match). If there is no `BUILD` keyword, this matching is attempted with the LHS of the CS rule (there should be at most one match). If that match fails, no TST node is created. If a TST node is created, as many elements on the build stack as there are principal children in the CS rule are popped off the build stack and attached in the appropriate order as children to the created node. That created node is them pushed onto the build stack. At the end of a successful parse, the build stack should contain a single element—the root of the TST representing the edited text. That root (and the implied TST) is the output of the edited-text parser.

We'll find it convenient in this chapter to use a fictitious form of parser we'll call the *component parser*. The component parser takes as input a fragment set, a CS rule set, an FS rule, an AS rule (with some indication as to the current principal child), and a string to parse; it gives as output some indication as to whether it accepted or rejected the input string. The component parser does not build a TST. Let S be a non-terminal unique to the input fragment set and CS rule set, S_{rules} be a set of CS rules, and P be the set of rules the component parser uses to parse its input text. S will be the start symbol for P. If the input fragment set is non-empty, then for each fragment f in the input fragment set, $S ::= f . \in S_{rules}$; and nothing else is in S_{rules}. If the input fragment set is empty, then for each input CS rule, r, whose LHS matches some element of the flattened LHS of the input FS rule, $S ::= r . \in S_{rules}$; and nothing else is in S_{rules}. In either case, P is the union of the input CS rule set and S_{rules}. P is used in the expected way to attempt recognition of the input string to parse. If the input string is recognized, the component parser emits some signal denoting acceptance; else, it emits some signal denoting rejection.

A *CS sentence fragment* is a string over the union of terminals and non-terminals of the language definition. We typically call CS sentence fragments simply *fragments* or *frags*.

Let c be an FS rule component. A *form of component formatter output* for c is a concatenation of zero or more of c's child components. A *form of component algorithm output* for c is a concatenation of fragment sets or the fragments from those sets, and possibly LDL symbols (like `::=` and `.`). The fragment sets are assumed to have been constructed as a consequence of executing the appropriate component algorithms on c's child components.

Definition 1 *A component algorithm for FS rule component c is* correct *iff, for all forms f of component formatter output for c, the component parser can use the component algorithm's output for c to parse f.*

Lemma 1 *Let c be an FS rule component. A component algorithm for c is correct iff there is a 1:1 and positional correspondence between the forms of component formatter output for c, and the forms of component algorithm output for c.*

Proof *A child component in the form of component formatter output means the component formatter will format that component child at that point in the formatting, and, of course, output the result of that formatting. A fragment in the form of component algorithm output will be used by the component parser to parse forms of component formatter output matching the contents of that fragment. A positional correspondence between an element of the form of component formatter output and an element of the form of component algorithm output means the parser will parse a subset of the component formatter's outputs for that element. If every fragment of the associated fragment set so corresponds, then the component parser will parse exactly the set of the component formatter's possible outputs for that element. If all elements so correspond, and neither side has elements the other side does not, then the component parser consumes no more and no fewer component children than the component formatter emits. Together with the defined behavior of the component formatter and component parser, this means the component parser accepts exactly those strings the component formatter emits.*

Theorem 1 *All component algorithms are correct.*

Proof *Follows from lemma 1 and the lemmas of correctness for each component algorithm.*

In the lemmas of correctness for each component algorithm, we show the positional correspondence between component children discussed in lemma 1 by numbering those children. We use ascending numbering for component children in successive positions. Sometimes, we

need to differentiate between different component children that may be emitted at the same position. In this case, we append a letter to the number.

Lemma 2 *A component algorithm for an FS rule component that's an alternation is correct iff it generates no CS rules and returns only its child's fragment set.*

Proof *Let* `AnyAlternation ::= Alt1 | Alt2 | ⋯ | Altn` *be an FS rule component that's an alternation of n alternatives. For* `AnyAlternation`*, the component formatter will output one of the alternatives:*

Alt_1
(1a)
Alt_2
(1b)
⋮
Alt_n
(1z)

Whenever the component formatter would emit Alt_1*, the generator algorithm will emit:*

$Alt_{1\,fragset}$
(1a)

Whenever the component formatter would emit Alt_2*, the generator algorithm will emit:*

$Alt_{2\,fragset}$
(1b)

And so on, up to the component formatter emitting Alt_n*, whereupon the generator will emit:*

$Alt_{n\,fragset}$
(1z)

So, by lemma 1, this component algorithm is correct.

Note: Here we refer to CS-style alternations, not the actual rule named `Alternation` (which, perhaps ironically, is actually a CS-style concatenation).

Lemma 3 *The component algorithms for* `FsRule`*,* `FsFixedRule`*,* `FsListRule`*,* `FsFixedChild`*,* `FsChild`*,* `IdentOrDontCare`*,* `FsSubtree`*,* `FsDefaultSubtree`*,*

FsHorizSubtree, *FsVertSubtree*, *FsFillSubtree*, *FsCondFmt*, *Formatter*, *Indenter*, *BolIndenter*, *ParentIndenter*, *PosIndenter*, *BolPlusIndenter*, *ParentPlusIndenter*, *PosPlusIndenter*, *BolMinusIndenter*, *ParentMinusIndenter*, *PosMinusIndenter*, *and* *CsAtom* *are correct.*

Proof *Every one of these FS rule components is an alternation, and every one of their associated algorithms generates no CS rules and returns only its child's fragment set. So, by lemma 2, these component algorithms are correct.*

Lemma 4 *A component algorithm for an FS rule component that's a simple concatenation is correct iff it generates no CS rules and returns a fragment set with a kind of Cartesian product of its children's fragment sets, where each element of the Cartesian product is a concatenation instead of an ordered set; the fragment set should also contain the empty string if the concatenation is defined by a star-list rule.*

Proof *We show the proof for the star-list version; the proof of the plus-list version is almost identical, except there is no correspondence between the empty list and the empty string. Let* `AnyStarConcat ::=* Elem .` *be an FS rule component that's a star-list type concatenation. The component formatter will emit this:*

$$\underset{(1a)}{Elem_1}\ \underset{(2)}{Elem_2}\ \cdots\ \underset{(n)}{Elem_n}$$

It's possible n = 0, in which case, the component formatter emits nothing, which we symbolize with the empty string:

$$\underset{(1b)}{\varepsilon}$$

The component algorithm emits this:

$$\underset{(1b)}{\varepsilon}$$

$$
\begin{array}{llll}
Elem_{1frag_1} & Elem_{2frag_1} & \cdots & Elem_{nfrag_1} \\
(1a) & (2) & & (n) \\
Elem_{1frag_1} & Elem_{2frag_1} & \cdots & Elem_{nfrag_2} \\
(1a) & (2) & & (n) \\
 & & \vdots & \\
Elem_{1frag_1} & Elem_{2frag_1} & \cdots & Elem_{nfrag_{|nfragset|}} \\
(1a) & (2) & & (n) \\
Elem_{1frag_1} & Elem_{2frag_2} & \cdots & Elem_{nfrag_1} \\
(1a) & (2) & & (n) \\
 & & \vdots & \\
Elem_{1frag_1} & Elem_{2frag_{|2fragset|}} & \cdots & Elem_{nfrag_{|nfragset|}} \\
(1a) & (2) & & (n) \\
 & & \vdots & \\
Elem_{1frag_2} & Elem_{2frag_1} & \cdots & Elem_{nfrag_1} \\
(1a) & (2) & & (n) \\
 & & \vdots & \\
Elem_{1frag_{|1fragset|}} & Elem_{2frag_{|2fragset|}} & \cdots & Elem_{nfrag_{|nfragset|}} \\
(1a) & (2) & & (n)
\end{array}
$$

So, by lemma 1, this component algorithm is correct.

Lemma 5 *The component algorithms for* `FsFixedChildList` *and* `FormatterList` *are correct.*

Proof *Both of these FS rule components are star-list style concatenations. Both their algorithms generate no CS rules and return a fragment set with an empty string and a kind of Cartesian product of their children's fragment sets, where each element of the Cartesian product is a concatenation instead of an ordered set. So, by lemma 4, these component algorithms are correct.*

Lemma 6 *The component algorithms for* `Alternation` *and* `IdentPlusList` *are correct.*

Proof *Both of these FS rule components are plus-list style concatenations. Both their algorithms generate no CS rules and return a fragment set with a kind of Cartesian product of their children's fragment sets, where each element of the Cartesian product is a concatenation instead of an ordered set. So, by lemma 4, these component algorithms are correct.*

Lemma 7 *The component algorithms for* `FsFixedDefaultRule`, `FsFixedHorizRule`, `FsFixedVertRule`, *and* `FsFixedFillRule` *are correct.*

Proof *We show only the proof on* `FsFixedHorizRule` *here; the other three FS rule components have nearly identical proofs. The FS rule component is as follows:*

```
IdentPlusList HORIZ -> FsFixedChildList .
```

In practice, this expands to the following FS components:

$$\underset{(1a)}{Ident_1}\ HORIZ \rightarrow \underset{(2)}{FsFixedChildList}\ .$$
$$\underset{(1b)}{Ident_2}\ HORIZ \rightarrow \underset{(2)}{FsFixedChildList}\ .$$
$$\vdots$$
$$\underset{(1z)}{Ident_n}\ HORIZ \rightarrow \underset{(2)}{FsFixedChildList}\ .$$

Here, $Ident_i$ *stands for the* i^{th} `Ident` *in flatten(*`IdentPlusList`*). The actual component formatter output is simply the* `FsFixedChildList`*. We show the correspondence with the FS rule itself to highlight the correspondence of LHSs. The component algorithm emits:*

$$\underset{(1a)}{IdentPlusList_{frag_1}} ::= \underset{(2)}{FsFixedChildList_{frag_1}}\ .$$
$$\underset{(1a)}{IdentPlusList_{frag_1}} ::= \underset{(2)}{FsFixedChildList_{frag_2}}\ .$$
$$\vdots$$
$$\underset{(1a)}{IdentPlusList_{frag_1}} ::= \underset{(2)}{FsFixedChildList_{frag_m}}\ .$$
$$\underset{(1b)}{IdentPlusList_{frag_2}} ::= \underset{(2)}{FsFixedChildList_{frag_1}}\ .$$
$$\vdots$$
$$\underset{(1z)}{IdentPlusList_{frag_n}} ::= \underset{(2)}{FsFixedChildList_{frag_m}}\ .$$

So, by lemma 1, this component algorithm is correct.

Lemma 8 *The component algorithms for* `FsListDefaultRule`, `FsListHorizRule`, `FsListVertRule`, *and* `FsListFillRule` *are correct.*

Proof *We show only the proof of* `FsListVertRule`*; the other three FS rule components have nearly identical proofs. There are two basic forms of* `FsListVertRule`*:*

```
IdentPlusList VERT ->* FsChild .
IdentPlusList VERT ->* FsChild || FormatterList .
```

We show only the proof of the second, more complex case; the proof of the first case is very similar. In practice, the second form expands to the following FS components:

$$\underset{(1a)}{Ident_1}\ VERT\ \text{->*}\ \underset{(2)}{FsChild}\ ||\ \underset{(3)}{FormatterList}\ .$$
$$\underset{(1b)}{Ident_2}\ VERT\ \text{->*}\ \underset{(2)}{FsChild}\ ||\ \underset{(3)}{FormatterList}\ .$$
$$\vdots$$
$$\underset{(1z)}{Ident_n}\ VERT\ \text{->*}\ \underset{(2)}{FsChild}\ ||\ \underset{(3)}{FormatterList}\ .$$

Here, $Ident_i$ *stands for the* i^{th} `Ident` *in flatten(*`IdentPlusList`*). The actual component formatter output is a series of* `FsChild` *components, separated by* `FormatterList` *components. We show the correspondence with the FS rule itself to highlight the correspondence of LHSs.*

The component algorithm may emit CS star-list rules (using the `::=*` *LDL symbol) or CS plus-list rules (using the* `::=+` *LDL symbol), depending on the* `FsListRule` *instance's associated* `AsListRule`*. Here, we show only the proof of the star-list rule version—the proof of the plus-list rule version is nearly identical.*

If $FormatterList_{fragset} \neq \{\varepsilon\}$ *, the component algorithm emits:*

$$\underset{(1a)}{IdentPlusList_{frag_1}}\ ::=*\ MgIdent\ ||\ \underset{(3)}{FormatterList_{frag_1}}\ .$$
$$\underset{(1b)}{IdentPlusList_{frag_2}}\ ::=*\ MgIdent\ ||\ \underset{(3)}{FormatterList_{frag_1}}\ .$$
$$\vdots$$
$$\underset{(1z)}{IdentPlusList_{frag_n}}\ ::=*\ MgIdent\ ||\ \underset{(3)}{FormatterList_{frag_1}}\ .$$

$$\begin{aligned} \texttt{MgIdent} &::= \underset{(2)}{\texttt{FsChild}_{\texttt{frag}_1}} \\ \texttt{MgIdent} &::= \underset{(2)}{\texttt{FsChild}_{\texttt{frag}_2}} \\ &\vdots \\ \texttt{MgIdent} &::= \underset{(2)}{\texttt{FsChild}_{\texttt{frag}_m}} \end{aligned}$$

Note that there will only ever be one element in $\texttt{FormatterList}_{\texttt{fragset}}$ *in this case, and it will always be an insertion token.*

If $\texttt{FormatterList}_{\texttt{fragset}}=\{\varepsilon\}$ *, the component algorithm outputs the same thing, except the star-list rules lack the* $\texttt{||FormatterList}_{\texttt{frag}_j}$ *part. Thus, there will be no correspondence to (3), but that's okay because the only time* $\texttt{FormatterList}_{\texttt{fragset}}=\{\varepsilon\}$ *is when the* `FormatterList` *has no insertion tokens or principal children (i.e., the formatter emits nothing but whitespace for it).*

So, by lemma 1, this component algorithm is correct.

Lemma 9 *The component algorithm for* `FsChildPlain` *is correct.*

Proof *In practice, the component formatter may emit one of four basic forms. In the special case of* `FsChildPlain`*, the component formatter will emit this:*

$$\underset{(1a)}{\texttt{String}}$$

When the corresponding `AsChild` *names its child type with a required or optional* `Ident`*, the component formatter may emit this:*

$$\underset{(1b)}{\texttt{Ident}}$$

When the corresponding `AsChild` *names its child type with a required or optional* `Alternation`*, the component formatter may emit this:*

$$\underset{(1c)}{\texttt{Alternation}}$$

When the corresponding `AsChild` *names its child type with an optional* `Ident` *or optional* `Alternation`*, the component formatter may emit nothing, which we symbolize as the empty string:*

$$\underset{(1d)}{\varepsilon}$$

For the special case of `FsChildPlain`*, the component algorithm will, due to class flattening, emit this:*

$$\underset{(1a)}{String_{frag_1}}$$

Note that `String`*'s fragment set only ever has one element. When the corresponding* `AsReqdChild` *names types with an* `Ident`*, the component algorithm will emit this:*

$$\underset{(1b)}{Ident_{frag_1}}$$
$$\underset{(1b)}{Ident_{frag_2}}$$
$$\vdots$$
$$\underset{(1b)}{Ident_{frag_n}}$$

When the corresponding `AsOptChild` *names types with an [* `Ident` *], the component algorithm will emit this:*

$$MgIdent_0 \texttt{ ::= } \underset{(1d)}{\varepsilon} \texttt{ .}$$

$$MgIdent_1 \texttt{ ::= } \underset{(1b)}{Ident_{frag_1}} \texttt{ .}$$
$$MgIdent_1 \texttt{ ::= } \underset{(1b)}{Ident_{frag_2}} \texttt{ .}$$
$$\vdots$$
$$MgIdent_1 \texttt{ ::= } \underset{(1b)}{Ident_{frag_n}} \texttt{ .}$$

$$MgIdent_0$$
$$MgIdent_1$$

When the corresponding `AsReqdChild` *names types with an* `{ Alternation }`*, the component algorithm will emit this:*

$$\underset{(1b)}{Alternation_{frag_1}}$$
$$\underset{(1b)}{Alternation_{frag_2}}$$
$$\vdots$$
$$\underset{(1b)}{Alternation_{frag_n}}$$

When the corresponding `AsOptChild` *names types with an* `[ { Alternation } ]`*, the component algorithm will emit this:*

$$MgIdent_0 ::= \underset{(1d)}{\varepsilon} .$$

$$MgIdent_1 ::= \underset{(1c)}{Alternation_{frag_1}} .$$
$$MgIdent_1 ::= \underset{(1c)}{Alternation_{frag_2}} .$$
$$\vdots$$
$$MgIdent_1 ::= \underset{(1c)}{Alternation_{frag_n}} .$$

$$MgIdent_0$$
$$MgIdent_1$$

So, by lemma 1, this component algorithm is correct.

Note that the `Idents` and `Alternations` used come from the corresponding `AsReqdChild` or `AsOptChild`, not the `FsChildPlain`. This is a necessary break from convention (normally, the FS rule component's own children would be used), for reasons outlined in the section on `FsChildPlain`'s algorithm.

Lemma 10 *The component algorithm for* `DontCare` *is correct.*

Proof *Technically, the component formatter will emit the form* `IdentPlusList`*. However, in our component algorithms we defer to* `FsChildPlain` *the recovery of that* `IdentPlusList`*. That is,* `DontCare`*'s component formatter output will be addressed by*

FsChildPlain's component algorithm, not by DontCare's. Thus, it is correct for DontCare's component algorithm to generate no CS rules and return $\{\varepsilon\}$.

Lemma 11 *The component algorithm for FsChildCase is correct.*

Proof *The component formatter will emit the FsFixedChildList of the first of FsCondFmtList's FsCondFmts that is satisfied, tested in order from leftmost to rightmost. If no FsCondFmt is satisfied, the component formatter will emit the FsFixedChildList associated with the ELSE clause. For the remainder of this proof, assume the ELSE clause of the FsChildCase, if there is one, is expanded as per the rules discussed in the FsChildCase section of chapter 2, and that the actual ELSE clause is disregarded.*

Note that, although each FsFixedChildList will have the same FsChild in it, the actual set of possible FsChildren the component formatter may emit for a given FsCondFmt will (except for the first FsCondFmt) be a subset of all possible emissions for the FsChild. The reason for this is, prior FsCondFmts will have already handled formatting certain subsets of the FsChild. To represent this, we introduce the two sets discussed in the FsChildCase section of chapter 2—Q and C.

The Q and C sets as discussed in chapter 2 were not true mathematical sets because they could be changed. In our proofs, we use a separate Q and C set for each FsCondFmt. Within the proofs of FsCondFmt types, we will subscript the two sets with the FsCondFmt number—i.e. the fifth FsCondFmt in the FsChildCase would have sets Q_5 *and* C_5. *For generality, we refer to the ith FsCondFmt. Note that* Q_0 *and* C_0 *are the "initial" Q and C sets for a FsChildCase.* Q_0 *and* C_0 *are initialized the same way for the component formatter as they*

are in `FsChildCase`*'s component algorithm. We assume there is some mechanism for communicating Q and C sets between* `FsChildCase`*s and* `FsCondFmt`*s.*

The component formatter emits

$$\underset{(1)}{\texttt{FsCondFmtList}}$$

The component algorithm emits

$$\underset{(1)}{\texttt{FsCondFmtList}_{fragset}}$$

So, by lemma 1, this component algorithm is correct.

Lemma 12 *The component algorithm for* `FsCondFmtList` *is correct.*

Proof *The component formatter will emit the* `FsFixedChildList` *of the first of its* `FsCondFmt`*s that is satisfied, tested in order from leftmost to rightmost. Note that some* `FsCondFmt`*s may be unsatisfiable—e.g., the second and successive instances of a duplicated predicate. Let us assume an arbitrary instance of* `FsFixedChildList` *has n satisfiable* `FsCondFmt`*s, that we do not count unsatisfiable* `FsCondFmt`*s at all, and that the* i^{th} `FsCondFmt` *matches. The component formatter will then emit this:*

$$\underset{(1)}{\texttt{FsCondFmt}_{i}}$$

Since $1 \le i \le n$ *, the component algorithm needs to emit fragments and rules capable of being used to parse any of those* `FsCondFmt`*s. The component algorithm emits this:*

$$\underset{(1)}{\texttt{FsCondFmt}_{1_{fragset}}}$$
$$\underset{(1)}{\texttt{FsCondFmt}_{2_{fragset}}}$$
$$\vdots$$
$$\underset{(1)}{\texttt{FsCondFmt}_{n_{fragset}}}$$

In the component algorithm's code, this is computed as a union of fragment sets. Unsatisfiable `FsCondFmt`*s will return empty sets, thus contributing nothing to the union. Thus, if* $\texttt{FsCondFmt}_{j_{fragset}}$ *(on the component algorithm side) is* $\texttt{FsCondFmt}_i$*'s (on the component formatter side) fragment set, then j = i.*

Since the `ELSE` *clause of* `FsChildCase` *is expanded into* `FsCondFmt`*s and appended to* `FsCondFmtList`*, at least one predicate must match (or else the* `FsChildCase` *is semantically invalid).*

So, by lemma 1, this component algorithm is correct.

Lemma 13 *It is correct for the component algorithms for the predicates—* `FsCondAbsent`*,* `FsCondPresent`*,* `FsCondMember`*,* `FsCondEmpty`*,* `FsCondNonempty`*,* `FsCondPlural`*,* `FsCondNonplural`*—to return the empty set if they're unsatisfiable.*

Proof *Within our framework, a component algorithm is incorrect if it outputs* $\emptyset$*, since then the component parser has nothing with which to build a grammar. However, for the purposes of this proof, a predicate never exists outside the context of some* `FsCondFmtList`*. The component algorithm for* `FsCondFmtList` *relies on the predicates' component algorithms to output* $\emptyset$ *whenever the respective predicate is unsatisfiable. This is to differentiate unsatisfiable predicates from satisfiable predicates that format to no insertion tokens or principal children (i.e., for which the component algorithm will output* $\{\varepsilon\}$*). Thus, it is permissible for predicates' component algorithms to output* $\emptyset$ *when the predicate is unsatisfiable.*

In the following proofs for predicates, let α, B, and γ represent the same things they do in chapter 2: For the `FsFixedChildList` associated with a predicate, let α be the

`FsFixedChild`ren before the format principal child, B be the format principal child being formatted, and γ be the `FsFixedChild`ren after the format principal child. α, B, and γ will be subscripted in ways that should be predictable by this point. The fragment sets for α and γ will only ever have one element since they'll be composed only of `String`s, `Indenter`s, and `LineBreak`s.

Lemma 14 *The component algorithm for* `FsCondAbsent` *is correct.*

Proof *On the component formatter side,* $Q_i = Q_{i-1} - \{0\}$. *If* $Q_{i-1} \ni 0$, *the component formatter emits*

$\underset{(1)}{\alpha_i}\ \underset{(2)}{\varepsilon}\ \underset{(3)}{\gamma_i}$

On the component algorithm side, $Q_i = Q_{i-1} - \{0\}$. *If* $Q_{i-1} \ni 0$, *the component algorithm emits*

MgChild `::=` $\underset{(2)}{\varepsilon}$ `.`

$\underset{(1)}{\alpha_{i_{frag_1}}}$ *MgChild* $\underset{(3)}{\gamma_{i_{frag_1}}}$

So, by lemma 1, this component algorithm is correct.

Lemma 15 *The component algorithm for* `FsCondPresent` *is correct.*

Proof *On the component formatter side,* $C_i = \emptyset$. *If* $C_{i-1} \neq \emptyset$, *the component formatter emits a* $\underset{(1)}{\alpha_i}$; *followed by a* $\underset{(2)}{B_i}$, *where* $B_i \in C_{i-1}$; *followed by a* $\underset{(3)}{\gamma_i}$.

On the component algorithm side, $C_i = \emptyset$. *If* $C_{i-1} \neq \emptyset$, *the component algorithm emits*

$\underset{(1)}{\alpha_{i_{frag_1}}}$ *MgChild* $\underset{(3)}{\gamma_{i_{frag_1}}}$

and, for each $c \in C_{i-1}$, *also emits a*

`MgChild ::=` $\underset{(2)}{c}$ `.`

So, by lemma 1, this component algorithm is correct.

Lemma 16 *The component algorithm for* `FsCondMember` *is correct.*

Proof *Let M be the* `Ident` *or* `Alternation` *following the* `MEMBER` *predicate, and* A_i *= flatten(M). Let* $T_i = A_i \cap C_{i-1}$. *On the component formatter side,* $C_i = C_{i-1} - T_i$. *If* $T_i \neq \emptyset$, *the component formatter emits a* $\underset{(1)}{\alpha_i}$; *followed by a* $\underset{(2)}{B_i}$, *where* $B_i \in T_i$; *followed by a* $\underset{(3)}{\gamma_i}$.

On the component algorithm side, $C_i = C_{i-1} - T_i$. *If* $T_i \neq \emptyset$, *the component algorithm emits*

$\underset{(1)}{\alpha_{i_{frag_1}}}$ `MgChild` $\underset{(3)}{\gamma_{i_{frag_1}}}$

and, for each $t \in T_i$, *also emits a*

`MgChild ::=` $\underset{(2)}{t}$ `.`

So, by lemma 1, this component algorithm is correct.

Lemma 17 *The component algorithm for* `FsCondEmpty` *is correct.*

Proof *On the component formatter side,* $Q_i = Q_{i-1} - \{0\}$. *If* $Q_{i-1} \ni 0$, *the component formatter emits*

$\underset{(1)}{\alpha_i}$ $\underset{(2)}{\varepsilon}$ $\underset{(3)}{\gamma_i}$

On the component algorithm side, $Q_i = Q_{i-1} - \{0\}$. *If* $Q_{i-1} \ni 0$, *the component algorithm emits*

`MgChild ::=` $\underset{(2)}{\varepsilon}$ `.`

$\underset{(1)}{\alpha_{i_{frag_1}}}$ `MgChild` $\underset{(3)}{\gamma_{i_{frag_1}}}$

So, by lemma 1, this component algorithm is correct.

Lemma 18 *The component algorithm for* `FsCondNonempty` *is correct.*

Proof *On the component formatter side,* $Q_i = Q_{i-1} - \{1, 2+\}$. *If* $Q_{i-1} \ni 1$, *the component formatter emits*

$$\underset{(1)}{\alpha_i} \quad \underset{(2)}{B_{i_{FsChild}}} \quad \underset{(4)}{\gamma_i}$$

If $Q_{i-1} \ni 2+$, *the component formatter emits a* $\underset{(1)}{\alpha_i}$, *followed by two or more* $\underset{(2)}{B_{i_{FsChild}}}$ *s separated from each other by* $\underset{(3)}{B_{i_{FsFormatterList}}}$ *s, followed by a* $\underset{(4)}{\gamma_i}$.

On the component algorithm side, $Q_i = Q_{i-1} - \{1, 2+\}$. *If* $Q_{i-1} \ni 1$ *or* $Q_{i-1} \ni 2+$, *the component algorithm emits one*

$$\underset{(1)}{\alpha_{i_{frag_1}}} \quad MgChild \quad \underset{(4)}{\gamma_{i_{frag_1}}}$$

If $Q_{i-1} \ni 1$, *then for each* $b \in flatten(B_i)$, *the component algorithm also emits a*

$$MgChild\ BUILD\ b\ ::=\ \underset{(2)}{b_{FsChild_{frag_1}}}\ .$$

If $Q_{i-1} \ni 2+$, *then for each* $b \in flatten(B_i)$, *the component algorithm also emits one of two possible outputs, depending on whether or not* $b_{FormatterList_{fraget}} = \{\varepsilon\}$. *We show the proof for the case where it does not, since that's the more complex one—the proof for the other case is very similar.*

$$MgChild\ BUILD\ b\ ::=++\ \underset{(2)}{b_{FsChild_{frag_1}}}\ ||\ \underset{(3)}{b_{FsFormatterList_{frag_1}}}\ .$$

So, by lemma 1, this component algorithm is correct.

Lemma 19 *The component algorithm for* `FsCondPlural` *is correct.*

Proof *On the component formatter side,* $Q_i = Q_{i-1} - \{2+\}$. *If* $Q_{i-1} \ni 2+$, *the component formatter emits a* $\underset{(1)}{\alpha_i}$, *followed by two or more* $\underset{(2)}{B_{i_{FsChild}}}$ *s separated from each other by* $\underset{(3)}{B_{i_{FsFormatterList}}}$ *s, followed by a* $\underset{(4)}{\gamma_i}$.

On the component algorithm side, $Q_i = Q_{i-1} - \{2+\}$. *If* $Q_{i-1} \ni 2+$, *then the component formatter emits one*

$$\underset{(1)}{\alpha_{i_{frag_1}}} \quad \texttt{MgChild} \quad \underset{(4)}{\gamma_{i_{frag_1}}}$$

and, for each $b \in flatten(B_i)$, *also emits one of two possible outputs, depending on whether or not* $b_{FormatterList_{fraget}} = \{\varepsilon\}$. *We show the proof for the case where it does not, since that's the more complex one—the proof for the other case is very similar.*

$$\texttt{MgChild BUILD } b \texttt{ ::=++ } \underset{(2)}{b_{FsChild_{frag_1}}} \texttt{ || } \underset{(3)}{b_{FsFormatterList_{frag_1}}} \texttt{ .}$$

So, by lemma 1, this component algorithm is correct.

Lemma 20 *The component algorithm for* `FsCondNonplural` *is correct.*

Proof *On the component formatter side,* $Q_i = Q_{i-1} - \{0, 1\}$. *If* $Q_{i-1} \ni 0$, *the component formatter emits*

$$\underset{(1)}{\alpha_i} \quad \underset{(2a)}{\varepsilon} \quad \underset{(3)}{\gamma_i}$$

If $Q_{i-1} \ni 1$, *the component formatter emits*

$$\underset{(1)}{\alpha_i} \quad \underset{(2b)}{B_{i_{FsChild}}} \quad \underset{(3)}{\gamma_i}$$

On the component algorithm side, $Q_i = Q_{i-1} - \{0, 1\}$. *If* $Q_{i-1} \ni 0$ *or* $Q_{i-1} \ni 1$, *the component algorithm emits one*

$\alpha_{\underset{(1)}{i_{frag_i}}}$ `MgChild` $\gamma_{\underset{(3)}{i_{frag_i}}}$

If $Q_{i-1} \ni 0$ *, the component algorithm also emits a*

`MgChild ::=` $\underset{(2a)}{\varepsilon}$.

If $Q_{i-1} \ni 1$ *, then for each* $b \in flatten(B_i)$ *, the component algorithm also emits a*

`MgChild BUILD b ::=` $b_{\underset{(2b)}{FsChild_{frag_i}}}$.

So, by lemma 1, this component algorithm is correct.

Lemma 21 *The component algorithm for* `Linebreak` *is correct.*

Proof *The component formatter emits whitespace for this, which we symbolize as the empty string:*

$\underset{(1)}{\varepsilon}$

The component algorithm emits this:

$\underset{(1)}{\varepsilon}$

So, by lemma 1, this component algorithm is correct.

Lemma 22 *The component algorithm for* `Ident` *is correct.*

Proof *In the FS, an* `Ident` *may be part of the LHS of a rule, the name of a principal child, the type of a principal child (which may or may not be a class), an alias for a* `String` *(i.e., function as an insertion token), or part of a* `MEMBER` *predicate. We're also concerned with* `Ident`*s used in the AS as types of principal children. In all cases,* `Ident`*'s component algorithm attempts to flatten the* `Ident`*. If the* `Ident` *refers to a class, the result will be the flattened class (which may be comprised of a single* `String`*, or one or more* `Ident`*s); else, the result will be the original* `Ident`*. Class flattening is needed when* `Ident`*s are used as LHSs*

(because CS rules cannot have classes as LHSs), and when `Idents` *are used with the* `MEMBER` *predicate.*

When the `Ident` *aliases a* `String`*, the proof of correctness is the same as for an actual* `String`*.*

In all other uses of `Ident`*, the component formatter does not directly emit anything. However, even in those cases we'll want the actual spellings of all members of the flattened* `Ident`*.* `Idents` *used in LHSs, for example, need to match up to the FS rule they were generated from, so that they can build the AST node that was formatted.* `Idents` *used a principal child types are needed so the parser will recurse on the same rules the formatter does.*

Lemma 23 *The component algorithm for* `String` *is correct.*

Proof *The component formatter will emit:*

```
String
  (1)
```

The component algorithm will emit this:

```
String
  (1)
```

So, by lemma 1, this component algorithm is correct.

Theorem 2 *Given an arbitrary AS rule set, A; FS rule set based on A, F; CS rule set generated from A and F, C; set of all possible ASTs rooted at any element of A, T; an arbitrary element t in T; the root of t,* t_{root}*; and an edited-text parser that uses C but the start symbol associated with* t_{root}*,* $p_{S=t_{root}}$*; if the tree formatter can use A and F to format t into e, then when* $p_{S=t_{root}}$ *parses e it will output t.*

Proof *By theorem 1, the semantic rules of LDL, and the behavior of the rule generator,* $p_{S=t_{root}}$ *can parse e. The rest of this proof is inductive. In both the base and inductive cases, as*

a consequence of applying the CS rule for t_{root}, $p_{S=t_{root}}$ *will build* t_{root} *(because the CS rule's LHS must be in the flattened LHS of some AS rule). This is the base case: If* t_{root} *has no children,* $p_{S=t_{root}}$ *will not pop anything off the build stack (there will be nothing to pop), and will push* t_{root} *onto the build stack. These are the inductive cases: If* t_{root} *has children that are not formatted with list predicates, if those children's ASTs were built correctly and pushed onto the build stack,* $p_{S=t_{root}}$ *will pop those children off the stack, attach them to* t_{root} *in the correct order, and push* t_{root} *onto the stack. If* t_{root} *has children that are formatted with list predicates, those children will be built correctly and pushed onto the stack by using their associated CS* `BUILD`*-rules; if the remaining children not so-formatted are built correctly and pushed onto the stack, then* $p_{S=t_{root}}$ *will pop all* t_{root}*'s children off the stack, attach them to* t_{root} *in the correct order, and push* t_{root} *onto the stack.*

Theorem 3 *Given an arbitrary AS rule set, A; FS rule set based on A, F; CS rule set generated from A and F, C; set of all possible ASTs rooted at the AST node associated with the defined language's start symbol, T; arbitrary element t in T; and edited-text parser that uses C and the defined language's start symbol, p; if the tree formatter can use A and F to format t into e, then when p parses e it will output t.*

Proof *This is a special case of theorem 2, where every instance of AST is rooted at the defined language's start symbol, instead of being rooted at any node type defined by A.*

Theorem 4 *Given an arbitrary AS rule set, A; FS rule set based on A, F; and CS rule set generated from A and F, C; if A and F are unambiguous, then C is unambiguous, too.*

Proof *Let t be an arbitrary TST rooted with its associated defined language's start symbol, p be the edited-text parser that uses C and the defined language's start symbol, and f be*

the tree formatter using A and F. If F and A are unambiguous, then f formats every TST conforming to A into a unique string. If C is unambiguous, then p parses every string output from f into a unique TST. By theorems 1 and 2, the p will parse any and only f's outputs; that is, the language output by f and the language recognized by p are identical. By theorem 3, p rebuilds any TST formatted by f; and by theorems 1 and 2, p builds no TST that cannot be formatted by f. Thus, if F and A are unambiguous, so too must be C.

CHAPTER 4

DISCUSSION

In this thesis we have presented a means of automatically generating a CS rule set from an FS rule set and AS rule set, and proven that means to be correct. Automating the generation of CS rules relieves language definers of much tedious, error prone work. In this chapter, we give a parting remark about the proofs in chapter 3, briefly discuss some post-processing that is done by Schütz on the generated CS rule set to prepare it for input into an LALR parser generator [5], discuss how Schütz handles redundant parentheses, mention LDL's precedence rule, and suggest directions for future research.

4.1 Remark about the proofs

While writing the proofs, several bugs in the component algorithms were revealed. For instance, for the component algorithms for list predicates, originally the algorithm treated B as just one type, whereas in practice, it may be a class of types. This necessitated adding a flattening operation to B in those component algorithms. Another bug was, originally, all predicates' algorithms were bracketed by the following code:

$F_{FsCondFmtList} := \{ \alpha$ `MgChild` $\gamma \}$
(predicate-specific code here... *)*
return $F_{FsCondFmtList}$

The problem with that was, were the precondition of a predicate's algorithm not met (i.e., were the predicate unsatisfiable), no CS rule would be generated for the `MgChild` referenced in the fragment! To fix this, we replaced the bracketing code with

$F_{FsCondFmtList} := \emptyset$
(predicate-specific code here... *)*
return $F_{FsCondFmtList}$

and set $F_{FsCondFmtList} := \{ \alpha$ `MgChild` $\gamma \}$ only when the predicate's precondition is met.

4.2 Post-processing

There's no analog in EBNF to LDL's construction of CS list rules. So, CS list rules need to be translated into equivalent CS fixed rules. There are many ways of doing this; we use a way that generalizes easily to star-rules, plus-rules, and plural-rules.

We generalize from an arbitrary CS star-list rule,

```
A ::=* B || C .
```

which we break into three cases: 1) the empty list,

```
A ::= .
```

2) the singleton list,

```
A ::= B .
```

and 3) the list with two or more elements:

```
A ::= BD .
D ::= CB .
D ::= CBD .
```

Here, `D` is a new non-terminal not used elsewhere in the CS rule set.

Were `A` a plus-rule, we would omit the empty list case. Were `A` a plural-rule, we would omit the empty list and singleton list cases. Were `A` to lack a separators list (i.e., have no `|| C` part), we would use this `D` rule, instead:

```
D ::= B .
D ::= BD .
```

Note that the choice of right recursion in the above rules was arbitrary. In Dr. Bates' implementation of Schütz, right recursion is used because the implementation builds nodes in lists in right-to-left order.

4.3 Redundant parentheses

Sometimes programmers use redundant parentheses in their code to emphasize the grouping of certain expressions, or simply because they're unfamiliar with the rules of precedence of the language they're using. For instance, a programmer may write something like this:

*(b * c) + (d * e)*

Assuming the language uses traditional rules of precedence for arithmetic operators, the two sets of parentheses are redundant—i.e., the same computation would result from that expression as from this one:

*b * c + d * e*

Typically in a compiler, when source code is parsed and an AST built, the AST will not record the use of redundant parentheses. This is fine, because the original source code does not need to be reconstructed from the AST. In the case of Schütz, however, the original source code does need to be so reconstructed, for reasons mentioned in chapter 1.

In order to rebuild source code from a TST, Schütz needs to remember all sets of redundant parentheses in the source code for which it was able to generate a TST (i.e., code that was syntactically legal). To do this, the language designer must add an extra AS rule/FS rule pair to represent a pair of parentheses. For instance, if the class `AsExpr` may have redundant parentheses, this AS rule/FS rule pair would be needed:

```
AsExprRePr :: AsExpr .
AsExprRePr -> "(" AsExpr ")" .
```

Note that there may be other forms of redundant bracketing that will require their own AS rule/FS rule pairs to deal with. For example, a language may allow optional bracketing around a single-statement then-body of an if-then statement.

We suspect LALR conflicts may arise out of this mechanism, but that's a problem outside the scope of this thesis.

4.4 LDL's precedence rule

LDL has a *precedence rule* that can be used to specify the precedence and associativity of CS rules in otherwise ambiguous grammars. Since our CS rule generator makes unambiguous CS rule sets given unambiguous FS rule sets and AS rule sets, we simply ignore the precedence rule, if present; and obviously have no need to create such a rule.

4.5 Directions for future research

It appears the AS rule set could be mechanically generated from the FS rule set were a few (non-backwards-compatible) changes made to LDL's FS rules. Perhaps the most obvious change is, the types of the format principal children on the RHS of an FS rule must be made to exactly match those of the corresponding abstract principal children. Currently, it's allowed for the types of format principal children to be a superset of those of their corresponding abstract principal children. Another obvious change is, the FS needs plus-list rules in addition to star-list rules. There are no doubt other, more subtle changes that would need to be made; once they were made, the process of mechanically generating a AS rule set from an FS rule set would be similar in spirit to that of mechanically generating the CS rule set.

REFERENCES

LIST OF REFERENCES

[1] Bates, R. M., "The Language Definition Language Ldl1, for the Schütz Semantic Editor," Computer Science Technical Report, Department of Computer Science, Wichita State University, Wichita, KS, June 2005.

[2] Bates, R. M., "The Language Definition Language for the Schütz Semantic Editor," Computer Science Technical Report, Department of Computer Science, Wichita State University, Wichita, KS, May 23, 2003.

[3] Bates, R. M., "Bootstrapping the Schütz Semantic Editor," Computer Science Technical Report, Department of Computer Science, Wichita State University, Wichita, KS, 2003.

[4] Bates, R. M., "Language Definition in the Schütz Semantic Editor," presented at JMLC2003, Joint Modular Languages Conference, Klagenfurt, Austria, published in LNCS 2789, pp 229-240, Springer Verlag, Berlin, 2003.

[5] Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers, Principles, Techniques, and Tools*, Addison Wesley Longman, Reading, Massachusetts, 1985, pp. 215-266.