ASCENT
Application-Specific Environment Transformer

David Garlan
Linxi Cai
Robert L. Nord[1]
Robert Stockton

December 1992
CMU-CS-92-180

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A shorter version of this paper appeared as "A Transformational Approach to Environment Specialization" in *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments* (December, 1992).

[1]Current address: Siemens Corporate Research, 755 College Road East, Princeton, New Jersey, 08540

This research was sponsored, in part, by the National Science Foundation under Grant Number CCR-9109469, and by Siemens Corporate Research, Inc. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government or of Siemens.

Keywords: Environment Generation, Program Transformation, Programming Environments, Application-specific Languages, Preprocessors, Application Generators, Reuse

Abstract

Current software development environments tend to lie at opposite ends of a spectrum: at one extreme are specialized application generators; at the other are general-purpose programming environments. The former provide strong support for system development and reuse, but are costly to build and available only for limited domains. The latter provide weak support, but are generally available and universally applicable.

We describe a technique for automating the production of application-specific environments that lie between these two extremes. The key idea is that these environments can be derived from general-purpose programming environments by a process of structured transformation. We describe a tool for performing these transformations. It provides a set of formal operators for specializing a description of a general-purpose environment; as operators are applied, the system semi-automatically builds a set of transformation rules used to translate application-specific programs into those of the original general-purpose environment. The output of the tool is an application-specific environment together with a program transformer that provides execution semantics for programs written in that environment.

Contents

1 Introduction **3**

2 Constructing Application-Specific Environments **5**
2.1 Current Approaches 6
2.1.1 Macros 6
2.1.2 Environment Generation 6
2.1.3 Preprocessors 7
2.2 A Transformational Approach 8

3 The Application-Specific Environment Transformer **10**
3.1 Components of ASCENT 11
3.1.1 Grammar Modification Operators 11
3.1.2 Transformation Rules 12
3.1.3 Automated Support 13
3.1.4 Program Translation 18
3.2 Relation to Existing Transformation Techniques 18

4 Case Studies **21**
4.1 A Dataflow Environment 21
4.1.1 Transformation Strategy 22
4.1.2 Grammar Operators and Automated Transformation 24
4.1.3 Transformation Table Editing 26
4.1.4 Auxiliary Processing 27
4.1.5 Summary 28
4.2 ARL 31
4.2.1 Transformation Strategy 31
4.2.2 Grammar Operators and Automated Transformation 33
4.2.3 Transformation Table Editing 34
4.2.4 Auxiliary Processing 38
4.2.5 Summary 41
4.3 Implicit Invocation for Ada 43
4.3.1 Ada Implementation 43

4.3.2 Transformation Challenges 45
4.3.3 Summary 47

5 Implementation **48**
5.1 ASCENT as a Language-Oriented Environment 49
5.2 Reusing the TransformGen Implementation 50

6 Assessment of ASCENT **52**

7 Related Work **56**
7.1 Reuse 56
7.2 Multi-Paradigm Environments 57

8 Conclusions **58**

A Operators and their Corresponding Rules **64**
A.1 Narrowing/Renaming 64
A.2 Restructuring. 65
A.3 Major Redefinition. 67

Chapter 1

Introduction

Software systems are rarely conceived in isolation. Instead, most systems represent a new instance of a product in some family of related systems, be they accounting systems, database products, or compilers. Each system typically shares with other applications in its family a common "framework" of behavioral and structural properties. An important goal of software engineering research is to improve software productivity by exploiting such frameworks to support software reuse.

Application generators, such as spreadsheets and report generators, do just that. These systems typically provide a specialized environment tailored to a particular application domain. A user enters a description of an application for that domain using a specialized, declarative notation; the environment then automatically generates an implementation of the application. Since these application descriptions are considerably simpler and shorter than the resulting system code, substantial increases in productivity arise [Cle88, HKN85].

Unfortunately, application generators have had relatively little impact on software engineering practice. There are at least two important reasons for this. First, few application domains are so clearly circumscribed that a simple, specialized notation is sufficient to describe all applications in that domain. For example, while applications in a particular domain may share a common framework, they may also need to exploit general-purpose computation to accomplish their task. In the case of compiler generators, for example, while programming language syntax can be easily described by a domain-specific notation such as context-free grammars, the semantic processing of a language is often defined using a general-purpose programming language.

Second, when an application generator does support general-purpose computing, the cost of building that generator is typically quite high—comparable to that of building a compiler for a new programming language. This is often too high a price to pay for a software support environment that is not intended to be marketed commercially.

As a result, current software development environments tend to lie at two ends of a spectrum: at one extreme are specialized application generators; at the other extreme are general-purpose programming environments. The former provide strong support for system development, but are available only for limited domains. The latter provide weak support,

but are generally available and universally applicable. What is needed is a way to produce application-specific environments that lie between these two extremes—and at low cost.

We describe an approach that provides automated assistance for constructing application-specific environments. The key idea is that an application-specific environment is obtained from a general-purpose environment by a process of structured transformation. This activity is carried out in the context of a development system that provides a set of formal operators for specializing a description of a general-purpose environment; as operators are applied, the system semi-automatically builds a set of transformation rules used to translate application-specific programs into those of the general-purpose environment. The system's outputs are an application-specific environment together with a program transformer that provides execution semantics for programs written in that environment.

We begin by considering how application-specific environments are typically implemented, and then contrast this with our approach. Then we describe our transformation-based development system. To illustrate the power and limitations of the techniques, we describe three non-trivial examples of application-specific environments that we have constructed. We briefly outline our current implementation and compare this work to related research. Finally, we evaluate the usefulness of our approach, and indicate areas where we believe improvements are possible.

Chapter 2

Constructing Application-Specific Environments

Consider the following family of software systems. For certain application domains, such as image or signal processing systems, application software is best modeled as a coarse-grained dataflow system. In this model the overall processing of an application is configured as a network of data transformations. When presented with new inputs, each node in the network produces new outputs that are in turn consumed by other "down-stream" nodes. The processing performed by an individual node can, in general, be an arbitrary computation. In an image processing application, for example, typical data transformations might include digitization, feature recognition, and image rendering. Figure 2.1 illustrates a signal processing application represented as a dataflow network. (See [DG90] for an industrial example.)

This family of applications defines an application-specific framework in which the software is organized as a collection of *filters*, which perform the individual computations, and a collection of *pipes*, which connect outputs of filters to inputs of other filters. The framework also includes an execution model in which filters are executed as inputs become available to them.

An application-specific environment for this domain would allow a programmer to define filters, pipes, and, for each filter, provide a block of action code that produces outputs from its inputs. Pipes and filters can be defined in a dataflow-specific (perhaps even graphical) notation. The action code, however, would be written in a suitable general-purpose programming language, subject to a number of constraints. For example, an action block might be constrained from accessing global variables, or restricted in its use of return values and parameters. As the programmer constructs an application, the environment would perform static checks on the dataflow "program" to ensure that it is well-formed. For example, it might guarantee that the dataflow graph is acyclic, that each input to a filter participates in a unique dataflow link, and that both ends of a link agree on the type of the data transmitted between filters. Finally, the environment would be able to build an executable version of the system.

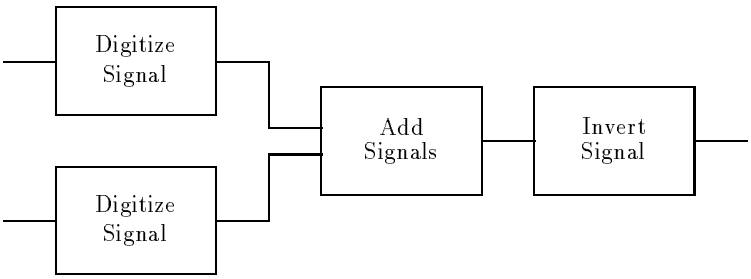

Figure 2.1: Addition of Digitized Waveforms

2.1 Current Approaches

Such an environment would typically be implemented in one of three ways: using macros, environment generation, or preprocessors.

2.1.1 Macros

The first approach is to define a set of macros on top of a general purpose programming language [KW87]. While macros are a relatively inexpensive technique for adding application-specific constructions to an existing language, they have a number of serious shortcomings. First, macros are primarily local and syntactic; they would have considerable difficulty in providing the global semantic checking needed for frameworks such as the one above. Second, macros are additive; they cannot be easily used to restrict the use of the language in which they are placed; such restrictions are often necessary to support the consistency of an application framework. Third, macros are optional; they rely on the goodwill of programmers to obey conventions and, in general, cannot be used to force the overall structure of a program into a particular form (such as filters and pipes from the example).

2.1.2 Environment Generation

The second approach is to build a new environment from scratch. This is done by defining a new language for the application domain, together with a special-purpose compiler or interpreter for that language. (Application generators adopt this approach.) Additionally, one might build a special editor for that notation, together with the underlying support for storing, retrieving, and composing programs constructed by it.

This approach achieves the goal of producing an environment appropriately tailored to the application, but it does so at high cost. Since a large portion of a general-purpose

programming language may be embedded in the framework, constructing the compiler would not be a trivial task. At best one might cannibalize an existing compiler. But connecting application-specific processing to the application-independent compiler can be complex, particularly when the framework interacts in non-trivial ways with the general-purpose code (as does the example above). Moreover, there is the added cost of managing program storage, producing a suitable interface to the user, and perhaps developing a special-purpose editor. Finally, there is the drawback that existing tools typically cannot be incorporated into the environment.

It is possible to reduce the high cost of developing such special-purpose environments through *environment generation* [RT84, HN86, DGHKL84]: a formal description of a programming language is given to an environment generator, which then produces an environment for that language. Usually the formal description consists of three parts: syntax, static semantics, and dynamic (or execution) semantics. Syntax is typically defined as a context free grammar. Productions in the grammar define the types of structures that can be manipulated by the environment. Static and dynamic semantics can be defined in a number of ways including attribute grammars [RTD83], logic programming [BGV90], natural semantics [BCD+88], or daemons [HN86, GM84]. The generated environment allows a user to create programs with a language-based editor, guarantees that those programs are syntactically and semantically valid, provides a run-time environment, and handles the storage and representation of programs as attributed abstract syntax trees.

Although environment generation decreases the effort required to build special-purpose environments, it too has problems. While the definition of syntax and static semantics is relatively straightforward, defining the execution semantics of a language remains a large stumbling block. In general, constructing this aspect of the environment is no less (and sometimes more) work than building the back end of a compiler. Again, it may be possible to use portions of an existing compiler, but this becomes more difficult as the special-purpose notation diverges from some existing language. Finally, environment generators do little to solve the problem of integrating existing tools: in general, tools must be written from scratch to take advantage of the special-purpose environment.

2.1.3 Preprocessors

The third approach is to define an application-specific language, and develop a preprocessor to translate between that language and a general-purpose one. This approach improves on macros because the new language can directly incorporate the application framework. Programmers are therefore forced to observe the conventions and constraints of the framework. Further, preprocessors allow reuse of existing tools that were designed to operate on general-purpose programs. In particular, this permits the reuse of a compiler for the general-purpose language.

However, preprocessors have three significant problems. First, the cost of developing preprocessors may be high, since they must be developed from scratch for each application area (although parser generators and text stream processors ease this burden). Second, preprocessors by themselves do not give the full advantages of a complete environment

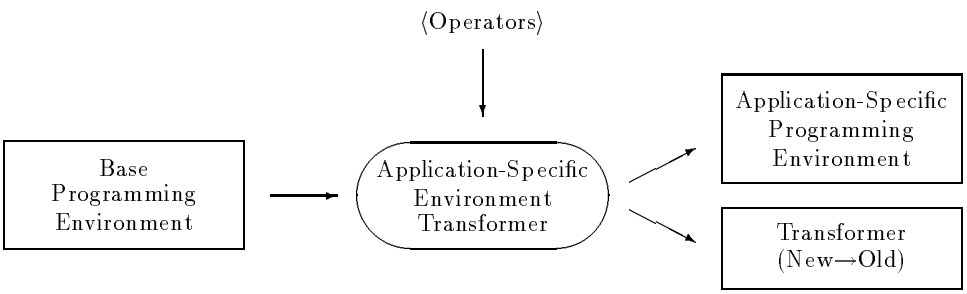

Figure 2.2: The Application-Specific Environment Transformer

tailored to the application. For example, preprocessors encourage batch processing, even when a set of interactive development tools exist for the general purpose language. Third, it may be difficult for users to relate the output of a tool (operating on the translated program) to the original program. For example, compiler error messages are typically associated with the translated program, and not the original source code.

2.2 A Transformational Approach

We have developed a different approach, one that builds on both the principles of environment generation and preprocessors, while solving many of their problems. Figure 2.2 illustrates the approach. Starting with a base environment that has been generated from a formal description, the implementor of an application-specific environment applies a sequence of transformation operators. These operators, described in detail later, modify the grammar of the base environment, thus producing an application-specific language. For instance, to obtain the example dataflow environment the implementor might add productions to describe filters and pipes, delete existing productions for declaring global data, and modify other productions so that the top-level structure of a program will consist of a set of filters and pipes. Additionally, the implementor would add new static semantic checks to the environment. (For example, the checks might guarantee that the dataflow network is well-formed—in the sense outlined earlier). When finished augmenting the base grammar, as described later, the implementor semi-automatically produces a transformer that can map abstract syntax trees of the new language to trees of the old.

This approach reduces the cost of producing application-specific environments in three significant ways. First, since the resulting environment is an incremental modification of an existing one, it allows the implementor to reuse the parts of the base environment that are shared by the new one. For environments that incorporate many features of a general-purpose environment the shared component can be quite large. Thus the cost in producing the new environment is roughly proportional to the size of the incremental

modifications, and not the size of the resulting environment. Second, since programs in the resulting environment are translated into those of the base environment for execution, as with preprocessors the implementor is able to use the execution semantics of the base environment. Third, for the same reason, other tools incorporated in the base environment can be applied to translated programs. While the output of these tools may be in terms of translated programs, as we will see, our approach can help alleviate problems of associating that output with the original source program.

Chapter 3

The Application-Specific Environment Transformer

What makes the transformational approach particularly attractive is the fact that we can construct a special-purpose system to guide and partially automate the process of producing application specific environments. The Application-SpecifiC ENvironment Transformer (ASCENT) is our implementation of such a system. ASCENT provides a set of formal operators for specializing the grammar of a base environment to create an application-specific environment. As the implementor applies the operators ASCENT automatically builds transformation rules to transform application-specific programs into those of the base language.[1] The implementor can augment these rules to handle aspects of the transformation that are not automated, again using formal operators provided by ASCENT. When the changes to the base environment are complete, ASCENT guarantees that the transformation rules are complete and consistent, builds a new environment for the application-specific language, together with a program transformer. This program transformer is automatically incorporated into the resulting application-specific environment, where it is used to create executable programs by translating them into those of the base environment.

Figure 3.1 provides a simple example of how ASCENT works. Production P1 with three components is transformed into production P2 with two components by applying two operators to the grammar and one to the resulting transformation rule. "Remove-component" removes the third component of P1, and "Exchange-components" reorders the remaining components. The resulting default transformation rule would specify that when the program transformer encounters an instance of production P2 it should transform it to one of P1, by reversing the two components of P2, and filling in an as-yet-to-be-determined value for the third component. To provide this value, the implementor would use the "Set-value" operator on the transformation rule to describe a tree construction in the language

[1] In this discussion, we distinguish between the transformations applied to the grammar of the base environment and the transformations applied to programs constructed by the new environment. We will consistently use the term "environment transformation *operators*" to refer to the former and the term "program transformation *rules*" to refer to the latter.

```
P1 ::= c1 c2 c3                    P2 ::= c2 c1

     Structure: P2
        Base: P1
        Component 1: Transform new component 2
        Component 2: Transform new component 1
        Component 3: INT-CONST Value = 0
```

Figure 3.1: Transformation Rule

of the base environment. This value is typically determined by the context of the program in the base environment: in this simple example, the implementor has simply set the value to the constant 0.

3.1 Components of ASCENT

ASCENT has four main components: (1) a set of grammar modification operators; (2) a transformation rule language; (3) automated support for developing transformations based on the operators applied to a grammar; and (4) run time support for mapping between application-specific programs and those of the base environment. We now consider each component. Afterwards, we explain how ASCENT leverages existing transformational techniques.

3.1.1 Grammar Modification Operators

As illustrated above, grammar modification operators are used to change a general purpose language into an application-specific one. The choice of these operators is crucial: the set of operators must be rich enough to easily achieve desired modifications, while at the same time they must be sufficiently constrained to permit formal manipulation and intuitive use.

Our choice of operators for ASCENT has been based on both theoretical and pragmatic concerns. ASCENT's operators have the formal property that they produce well-formed grammars, and that they compose with each other. On the practical side, much of our effort has focused on using real examples to understand what set of operators are sufficient to make the approach practical, and for what classes of application-specific environments these operators are relevant. Based on these case studies, three of which are described in Chapter 4, we have found it useful to group operators into three broad categories.

- **Narrowing/Renaming.**

  These include deleting a production from a grammar; deleting a production from a set of choices on the right hand side of a production; renaming a production to reflect the new application; and making a copy of a production (as a prelude to further

specialization).[2] Since the resulting grammar is effectively a subset of the original (subject to renaming), trees constructed with the application-specific environment will be valid trees in the base environment. Hence only trivial transformations are needed to handle these changes.

- **Restructuring.**

  These include: deleting a component of a production; reordering components; changing a list component to fixed arity; duplicating a variable arity component; replacing a component by its children; replacing a group of components by a new production whose children are the group of components; and changing a terminal to a nonterminal, and vice versa. These operators handle two kinds of situations: (1) when application-specific programs provide the same kinds of information as the base environment, but are structured in a different way (*e.g.*, reversed components, as above); and (2) when a variable portion of the base environment is implicitly defined in the application environment (*e.g.*, the value of a deleted component, as above).

- **Major Redefinition.**

  These include: adding a production to a grammar; adding a component to a production; and adding a production to the set of choices on the right hand side of a production. These operators are used to add new productions and components to a grammar. They arise when there is no direct analogue in the base language for the new construct. Since there is no corresponding production in the base grammar the implementor must provide the translation for this new production by directly editing the rule tables.

In practice we have found that an application-specific environment is most easily developed by applying these classes of operators in order of difficulty. First, the base grammar is narrowed by eliminating constructs that will not be needed in the new environment. The remaining productions are then renamed and locally restructured to fit the application domain. Finally, productions that have no counterpart in the base environment are added to the application-specific environment.

3.1.2 Transformation Rules

The program transformation rules are organized in a table: for each production in the new environment's grammar, an entry in the table defines a rule for translating instances of that production into instances of productions in the base environment. Each rule prescribes (a) what base environment structures should be substituted for the application-specific one, and (b) how the components of those structures (if any) should be recursively transformed.

[2] In this discussion we assume that non-terminal grammar productions are of the form "P ::= A B C", "P ::= seq of A", or "P ::= A | B | C". We refer to the first case as a *fixed arity production* in which A, B, and C are *components* of P. We refer to the second case as a *variable arity production*. We refer to the third case as a *union production* and the elements of the union as *choices*.

Initially the table contains the identity transformation for all productions. As operators are applied to the base environment grammar, rules are augmented to compensate for that change. See Appendix A for a list of the operators and their corresponding rules.

Our earlier example (Figure 3.1) illustrated a particularly simple example; in general the transformations may involve more complex reconstruction. For example, as we will see in Section 4, transformations can be nested, and the component constructions in the rule table support context-sensitive restructuring through the use of "tree expressions" to obtain their values.

The effect of using a tree expression is to replace the value of the component by a subtree returned by the tree expression. The first component in Figure 3.1, could, for example, have been specified by "Transform new parent, component 1, component 2", which would specify the second component of the first component of the source node's "grandfather". (This might or might not be the same node specified by simply "component 2".)

Selection of an appropriate transformation rule may be made conditional (on the transformation context) by invoking an implementor provided "conditional function." ASCENT also permits a component entry to contain an arbitrary function that returns a new structure to be placed in that entry. For example one might use a conditional function to specify the third component in terms of a tree construction or the subtree defined by the tree function compute-subtree.

```
if do-prune() then INT-CONST
                       Value: ''0''
                  else Replace with tree defined by compute-subtree()
```

Although rarely used, this serves as an escape hatch for transformations that are outside the scope of ASCENT.

Taken together, these options allow the implementor to specify simple transformations in a simple declarative manner, while providing sufficient power to allow arbitrarily complex transformations to be performed.

3.1.3 Automated Support

Thus far we have seen that ASCENT provides automated support by supplying a well-defined set of operators, and by partially automating the construction of program transformation rules. For those cases when the transformation rules cannot be provided or are not sufficient, ASCENT allows the implementor to edit the rule table using rule-changing operators.

In addition, ASCENT guarantees that transformation tables are complete and consistent before the implementor can generate an application-specific environment. In this context *completeness* means that every production in the application-specific environment has some translation and that any missing components are given values. *Consistency* means that transformation rules always result in the creation of a syntactically legal tree in the base language.

```
P1 = c1 c2 c3
C2 = s1 s2

        Structure:  P1
              Base:  P1
              Component 1:   Transform new component 1
              Component 2:   Transform new component 2
              Component 3:   Transform new component 3

        Structure:  C2
              Base:  C2
              Component 1:   Transform new component 1
              Component 2:   Transform new component 2
```

Figure 3.2: Before the Lift operation

Grammar Modification Operators

As described above, a new environment is created by modifying the base environment into the desired form, while simultaneously maintaining a set of transformation rules to convert a program tree back into the base grammar. In this section, we describe the facilities which allow these two processes to be performed in parallel, while maintaining consistency between the grammars and the transformation rules.

Much of the task of restructuring the base grammar can be performed simply by using a few operators which are specially designed to ease common tasks within ASCENT. Because these are high level operations with a full understanding of the ASCENT environment, they are able to freely examine and update all available information on the grammar. This allows us to provide a very simple interface which nonetheless can make complex changes to both the grammar and the transformation rules. Here, we describe a few representative examples.

Lift. The "Lift" operation replaces a single component of a production with the set of components from some production that would be legal for the original component. In effect it "lifts" the contents of a node's child so that they are added to the content of the parent node. The contents of the "lifted" production are unchanged.

Because the system has complete knowledge of the grammar, the lift operation can determine which productions are legal at the component being lifted, and thus which productions are legal candidates for lifting. If there is only one possibility, it is automatically chosen. If there are several, then the user is presented with a menu of possibilities from which to choose. This is used to collapse levels of hierarchy in a language.

The system examines the production being lifted to determine its components and lifts these into place in the target production. If the lifted production is a list, then the user is given the opportunity to choose the number of elements that should be lifted.

```
P1 = c1 s1 s2 c3
C2 = s1 s2

        Structure:  P1
              Base:  P1
              Component 1:   Transform new component 1
              Component 2:   C2
                    Component 1:  Transform new component 2
                    Component 2:  Transform new component 3
              Component 3:   Transform new component 4

        Structure:  C2
              Base:  C2
              Component 1:   Transform new component 1
              Component 2:   Transform new component 2
```

Figure 3.3: After "Lift c2"

For example, consider applying lift to the second component in Figure 3.2. Once the structure of the production has been modified, the system is left with the task of updating the transformation rule for the target production to reflect the changes to the production (Figure 3.3). This involves two different activities: First, references to components in the production must be renumbered to reflect the modified positions of some of the components (such as component c3 in the above figure, which moves from being component 3 to being component 4). Secondly, it must replace references to the lifted component with code to construct the production in the base grammar corresponding to the lifted production. (It does this by copying the transformation rule from the lifted production and renumbering it according to the components position in the target node.) Since the transformation rule may contain several levels of construction, the system must recursively walk the entire transformation rule to find component references which must be renumbered or replaced.

Lower. The "Lower" operation is the logical inverse of "Lift". It takes a sequence of components in a production and replaces them with a single component which designates a new production containing the "lowered" components. All of the information contained in the original production will be retained, but it will be spread out across two "levels" of the new tree. This is used to introduce additional layers of hierarchy.

This operation must create a new grammar production corresponding to the new grammatical layer. Since there is no obvious method of automatically naming these new entities, the operation prompts the user for names. However, ASCENT checks through existing production names to insure that the new names are not already in use.

The process of creating the new production and transferring the "lowered" components to the new production is tedious, but straightforward. The user is left with the task of filling in several fields of the new production which cannot be determined by the system.

```
P1 = c1 c2 c3 c4

        Structure:  P1
              Base:  P1
              Component 1:   Transform new component 1
              Component 2:   Transform new component 2
              Component 3:   Transform new component 3
              Component 4:   Transform new component 4
```

Figure 3.4: Before the Lower operation

```
P1 = c1 s1 c4
S1 = c2 c3

        Structure:  P1
              Base:  P1
              Component 1:   Transform new component 1
              Component 2:   Transform new component 2, component 1
              Component 3:   Transform new component 2, component 2
              Component 4:   Transform new component 3
```

Figure 3.5: After "Lower c2-c3"

Take, for example, lowering the second and third component in Figure 3.4. The result is show in Figure 3.5. The process of updating the transformation rule is similar to the one employed for the "lift" operation. Again, component references must be renumbered to account for components which have been retained but changed position. In addition, references to lowered components must be modified to refer to components of the new production, which involves adding an extra element to the path expressions and renumbering the sub-components.

Make-terminal and Make-nonterminal. These operations change the general structure of a production. As much of the node's description as possible is retained, but there is typically not enough similarity in structure for a meaningful transformation rule to be constructed. The best solution is therefore to leave the parts of the transformation rule empty so that the user will be required to fill them in. (ASCENT guarantees that all such rules are completed before a new environment is generated.)

Language-Oriented Editing Operators

Users may also modify the grammar specification directly using lower level editing operations. These operations are less powerful than the ones described above, but they are potentially more versatile, since they are not based upon any particular assumptions concerning the user's intentions. Typical operations which will affect the structure of the grammar (and thus have a potential effect on the transformation process as well) include addition, deletion, and reorganization of components, attributes, or productions.

Appropriate updates to the transformation rules are, unfortunately, both less useful and more difficult to implement for the basic editing operations. The primary difficulty arises from the fact that an editing operation may have many significantly different effects upon the grammar structure depending upon the context in which it is applied. The "delete" operation, for example may be applied to a component (or to all components), an attribute (or simply the attribute type), a value specification, or an entire production. All of these can introduce significant changes in the grammar, and should therefore trigger corresponding changes in the transformation rules to ensure that they remain consistent.

Maintaining consistency is similar for most of the lower level operations It typically involves renumbering or deleting component references (or renaming attribute references) when component or attribute lists are modified, or removing entire transformation rules when a production is deleted.

Unfortunately, because these are low level "primitive" editing operations, it is not always possible to infer the user's intent. For example, if the user were to remove a component and then add an identically named component elsewhere, the system would be unable to infer that the component was actually being moved to a different place. Instead, it would simply delete the reference to the original component and leave it blank when the new component was added. A higher level operation, such as "move component", on the other hand, would directly convey the user's broader intentions.

Modifying the Rule Tables

Finally, the user can explicitly modify the transformation rules without making any corresponding change to the grammar itself. (Figure 3.1 illustrated this when a constant value of 0 was inserted into a grammar rule.) The same basic editing operations that are available for the grammar may be applied to the transformation rules to produce arbitrarily complex transformations. This may be necessary if there is no close correspondence between the application-specific grammar and the base grammar, or if the grammar was modified in such a way that the system could not automatically infer the correct transformations. Moreover, explicit modification is used for transformations that require conditionals and tree functions, since these are not typically produced by any of the automated rule creation mechanisms.

3.1.4 Program Translation

When the environment implementor has finished applying operators and developing transformation rules, ASCENT generates an application-specific environment for the new language using standard environment generation techniques. However, in addition to the typical facilities provided by a generated environment, the resulting environment contains a program transformer.

The transformer can be invoked by any tool within the application-specific environment. It functions as an interpreter of the transformation tables. To convert an application-specific program to one in the base environment the transformer starts at the root of the program and transforms it according to the transformation rules to produce the root of a base environment program. It then recursively transforms the components of the newly created root, as directed by its transformation rule. The resulting tree can then be read and manipulated by tools from the original base environment. In particular, it can be executed using whatever execution engine was provided in the base environment.

An important feature of the translation process is that as the transformer performs the tree transformation, it annotates the translated program with attributes that indicate where each node came from in the original program. This information can be used by tools to interact with the user in terms of the original program. As a simple example, if a debugger stops at some node in the translated tree, the cursor can be displayed at the corresponding node of the original tree.

3.2 Relation to Existing Transformation Techniques

ASCENT builds strongly on existing transformational techniques for maintaining software development environments [Bal85, SZ90, LH90], where the primary concern is the problem of maintaining persistent data (*e.g.*, programs) in the presence of changes in data format. Of these efforts the most closely related is the TransformGensystem [GKS86]. Like ASCENT, TransformGen takes as input a grammar for a base environment, allows an implementor to apply operations to obtain a new grammar, and then generates a new

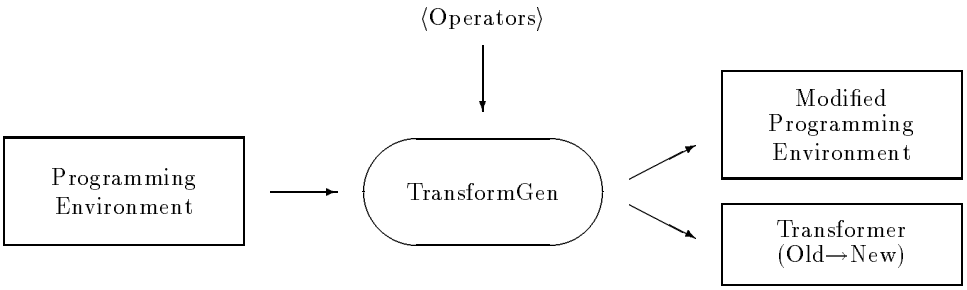

Figure 3.6: TransformGen

environment together with a program transformer. (See Figure 3.6.) Also, as with ASCENT, the resulting program transformer is driven by program transformation rules that are semi-automatically derived from the operations performed on the base environment.

However, since TransformGen (and similar systems) was developed to solve a completely different problem, its transformers are used in quite a different way. Instead of translating programs from the new (application-specific) environment into the old (base) environment, it is used to convert old programs into a format that is syntactically valid in the new environment.

Despite this difference there are several aspects of TransformGen that we were able to leverage in the development of ASCENT.

- **Operators.** Many of the grammar operators are the same. However, we found it necessary to introduce a richer set of restructuring operators to handle the more complex grammar transformations that arise in the definition of frameworks. We will see in the case study an example of the "Duplicate-var-arity" operator that produces two components from a single variable arity component. This has two important advantages over using more primitive TransformGen operators. The operator corresponds to the restructuring that the implementor has in mind, and the transformation rule that merges the two components back into a single component is automatically produced.

- **Transformation Rules.** The transformation rule language carries over. However, the automatic creation of table entries is quite different, since the transformer works in the reverse direction. For example, removing a production is easy for ASCENT, but difficult for TransformGen, while the opposite is true of adding a production.

- **Automated Support.** Consistency and completeness checks carry over. However, ASCENT must be more strict to guarantee that the translation of any complete application-specific program results in a complete base language program. In contrast, it is acceptable for TransformGen to leave some parts of a transformed tree uninstantiated, since the user can simply fill in the missing parts after the program has been transformed.

- **Program Translation.** Translation proceeds similarly in both systems. However, in ASCENT the transformer is a resident tool within the resulting environment. In contrast, a TransformGen translator is not an integral part of an environment, since it is only (implicitly) invoked when an out-of-date program is read into a new environment.

Chapter 4

Case Studies

We now consider three case studies to illustrate both the benefits and limitations of the approach. In the case studies we created an application-specific environment through a process of structured transformation on an existing general-purpose programming environment. In the first case study, we created an environment for the dataflow framework introduced earlier in Section 2. In the second case study, we created an environment for ARL, a special-purpose language for manipulating abstract syntax trees. In the third case study, we added an implicit invocation mechanism to Ada.

4.1 A Dataflow Environment

An application-specific environment for the dataflow domain would allow a programmer to define software as a collection of filters and pipes. Filters transform streams of data from a set of inputs to a set of outputs. Pipes connect the outputs of one filter to the inputs of others.

A simple pipe and filter program is illustrated graphically in Figure 4.1. This program changes a stream of characters so that the characters alternate between upper and lower case. This is accomplished by defining four filters and linking them together appropriately using pipes. The program reads the standard input into the split filter, which alternates

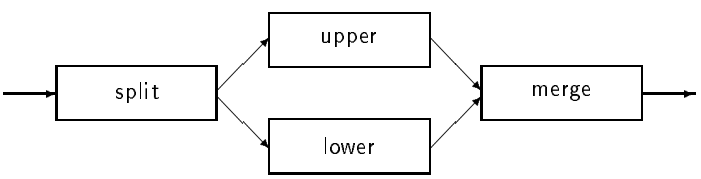

Figure 4.1: Alternating Case Example

```
Dataflow AlternatingCase:

Filters:

  filter: split
     inputs: char in
     outputs: char left, right
     static vars: bool l
     init: l = true;
     action:
         if l then
           write(left,read(in));
         else
           write(right,read(in));
         endif;
         l = not l;
 ...

Pipes:

   stdin -> split:in
   split:left -> upper:in
   split:right -> lower:in
   upper:out -> merge:left
   lower:out -> merge:right
   merge:out -> stdout

end AlternatingCase.
```

Figure 4.2: Alternating Case

sending output to the filters, upper and lower, which change the case of the character appropriately. The outputs from upper and lower are then sent to the merge filter which combines the original character stream. In Figure 4.2 we show the filter definition for split and the connections between the filters. The other filter definitions have been elided.

4.1.1 Transformation Strategy

Our goal is produce an application-specific pipe and filter (PF) environment from an existing one for the C language. Using ASCENT we will do this by transforming the C grammar to produce one specialized for this application family. In particular we can describe filters by specializing C procedures. This allows us to reuse the part of the C language for type declarations, statements, and expressions (perhaps with minor renaming). In the remainder of this section we examine how this is done. First we show how ASCENT operators are applied to the grammar to produce a specialized language and an associated set of transformation rules. Next we describe the additional operations required on those transformation tables.

```
Terminals:
   FIDENT = {usesite}                    FNAMEDEF = {defsite}

Nonterminals:
   PROGRAM ::= <extdef>                  DFLOW ::= dname filters connections
                                         FILTERS ::= <filter>

   PROCDEF ::= otype procname            FILTER ::= filtername inputs outputs
               lparam body                          decls action action

   DECLCOMPOUND ::= ldecl lstat          ACTION_LANGUAGE ::= decls statements

   DECLS = <declaration>                  DECLS = <decl>
   STATS = <stat>                         STATEMENTS = <statement>

   STATEMENT = statement

                                         CONNECTIONS ::= <connection>
                                         CONNECTION ::= portid portid

Classes:
                                         dname ::= DNAME
   extdef ::= EXTPROC PROCDEF ...
                                         filters ::= FILTERS

   otype ::= EMPTY INT CHAR ...

                                         filter ::= FILTER
   procname ::= FIDENT ...               filtername ::= FNAMEDEF

   lparam ::= LPARAM                     inputs ::= INPUTS
                                         outputs ::= OUTPUTS

   body ::= DECLCOMPOUND                 action ::= ACTION_LANGUAGE

   ldecl = DECLS                         decls = DECLS
   lstat = STATS                         statements = STATEMENTS

   declaration = OBJDEF ...              decl = DECL

   stat = STATEMENT IF ...               statement = ASSIGNMENT IF WRITE
   statement = ASSIG PROCCALL ...

                                         connections ::= CONNECTIONS
                                         connection = CONNECTION
                                         portid = PORTID STDIN STDOUT
```

Figure 4.3: C and Dataflow Grammar

```
PROCCALL = procexp lexp                READ = port

                Structure: READ
                     Base: PROCCALL
                     Component 1:   FIDENT
                           Value:   "READ"
                     Component 2: LEXP
                           Component 1: Transform new component 1
```

Figure 4.4: Deleting a Component

Finally, we detail the kinds of auxiliary processing needed to complete the PF environment.

4.1.2 Grammar Operators and Automated Transformation

There are two main aspects of C that must be specialized. The first is the "action" sublanguage that determines what kinds of computations filters can perform. By and large, this is C. However, we need to build in special notation for reading and writing to pipes. For illustration we also show how to modify the conditional statement of C. The second area of specialization is the declaration of filters. As we have indicated, filter declarations are like C procedure declarations. However, the usual notion of formal parameters must be replaced by the notion of pipe inputs and outputs.

Action Language. Figure 4.4 shows how we change a C description of a procedure call into a read operation in the PF action language. The parameter component is changed to *fixed arity* and *renamed*, and the procedure expression component is *deleted*. The transformation rule that is generated requires the user to fill in the information associated with the deleted procedure expression component: here, we inserted the procedure name READ.

The restructuring of the second example shown in Figure 4.5 is more complex. We change a C description of a conditional statement into a corresponding PF conditional statement. The former allows a single statement in each branch of the conditional, the latter allows for a sequence of statements. Examining the C single-statement class definition, we see that one choice available is a compound statement, which in turn is a list of declarations and statements. It is this list of statements, lstat, which match the PF statements. The C definition of the conditional statement has more structure than is necessary for the corresponding PF statement. Using the *lift* operator to destructure singlestat in this context into statements will automatically produce the transformation rules.

Filter Definition. In Figure 4.6, we show the result of transforming a C grammar procedure definition into a PF filter after applying many different kinds of operators. We *deleted* the type of the procedure and edited the transformation rules so that a filter implementation has the void type (this is similar to the example seen in Figure 3.1). The procedure name

```
IF = exp singlestat oelse                IF = bool_expr statements statements
singlestat = STATEMENT COMPOUND2 ...
COMPOUND2 = ldecl lstat

               Structure:  IF
                     Base:  IF
                     Component 1:   Transform new component 1
                     Component 2:   COMPOUND2
                           Component 1:   DECLS
                                 List:   Skip
                           Component 2:   STATS
                                 List:   Transform new component 2
                     Component 3:   ...
```

Figure 4.5: Lifting a Component

```
   PROCDEF ::= otype procname params    FILTER ::= filtername inputs outputs
                 body                                        decls action action

               Structure: FILTER
                    Base: PROCDEF
                    Component 1: VOID
                    Component 2: Transform new component 1
                    Component 3: List: Transform new component 2,
                                       Transform new component 3
                    Component 4: DECLCOMPOUND
                       Component 1 : DECLS
                          Component 1: Transform new component 4
                       Component 2 : STATS
                          Component 1: Transform new component 5
                          Component 2: Transform new component 6
```

Figure 4.6: Restructuring a Production

```
PROCCALL = procexp exps                FILTER = fnamedef inputs outputs ...

    Structure: FILTER
        if defn_phase() then
           Base: PROCDEF
              ...
        else
           Base: PROCCALL
           Component 1: Transform new component 1
           Component 2: List
                        Transform new component 2
                        Transform new component 3
```

Figure 4.7: Many-to-one Grammar Transformations

is *renamed* to the filter name. The next two changes are forms of restructuring. The list of parameters is *duplicated* to model the input and output ports of the filter. The corresponding transformation rule defines how to combine the inputs and outputs by merging the two parameter lists together. The body of the procedure definition is destructured by *lifting* the components of the procedure body to reveal declarations and statements and *duplicating* the statements. These are then *renamed* to model the PF declarations and action routines. The corresponding transformation rule defines how to combine the declaration and action routines by instantiating the compound statement production that they were derived from.

While procedure definitions can be used to model filters, the description of our special purpose environment is not yet complete. We have not dealt with the issue of executing the filters nor of how they are linked to the pipes. For example, in Figure 4.9, part of the execution of filters and pipes is contained in the dup and close statements that initialize the input and output ports. Since there is no structure in our base language corresponding to pipes (*e.g.*, as procedures definitions correspond to filters), ASCENT operators are not as useful for this part of the PF translation.

4.1.3 Transformation Table Editing

Filter Execution. The filter execution is modelled as procedure invocation in a separate process. Ideally, we would like to start with a C procedure call and apply the grammar transformation operators to *rename* the procedure name to the filter name, *destructure* the parameter list into the inputs and outputs of the filter while ignoring the rest of the filter definition (such as the state and action language). However, we have already constructed the definition of a filter from a procedure declaration in the previous section. We need a special grammar operator to associate more than one production in the base language with a single production in the application-specific language. We achieve this functionality by editing the transformation rule table directly so that a filter production produces the procedure definition and procedure call in separate passes (Figure 4.7).

4.1.4 Auxiliary Processing

To complete our environment we must provide a way to handle the execution semantics of pipes.

Pipe Definition. A pipe data structure is defined for the pipes and filters program. Here the collection of pipes is represented as an array, where each pipe is a pair of file descriptors for the input and output to the pipe. This data structure is fixed with the exception of the number of pipes "@X", which must be calculated from the connections declaration.

```
#define NUM_CONS @X
int fildes[NUM_CONS][2];
```

We edit the rule table to construct the code for the pipe data structure, filling in the number of pipes by writing a tree function that computes the number of pipes from the connections declaration. As with much of auxiliary processing this function is only a few lines long.

Pipe Execution. The pipes are implemented as a UNIX pipe call. In addition, the pipe file-descriptor pairs are given descriptive names derived from the names of the filters where they are defined, and their formal parameter names.

Standard input and output are treated separately as special cases.

```
#define split_INP_in 0                          stdin -> split:in
```

In the general case, the data structure is constructed from the connection declaration, "@X" is dynamically calculated and is the position assigned to the pipe in the file descriptor (fildes) array.

```
pipe(fildes[@X]);                               split:left -> upper:in
#define upper_INP_in fildes[@X][0]
#define split_OUTP_left fildes[@X][1]
```

As is the case with the pipe definition, we edit the rule table to construct the appropriate code, and use tree functions (again, only a few lines long) to construct the information that must be dynamically calculated.

Main Program. To define the top-level dataflow program, we change the C description of a program into a PF description of the top-level system. The production is renamed appropriately and changed from a variable arity to a fixed arity production of two components. The two components will be definitions of the filters and pipes. We need to edit the rule table in this case because the filters and pipes will be translated into different program segments (one for definition and one for execution) in the base programming language.

```
PROGRAM = <extdef>                    DFLOW = dname filters connections

            Structure: DFLOW
               Base: PROGRAM
               List: /* define pipe data structure <<component 3>> */,
                     /* filter definitions <<component 2>> */ ,
                     /* main program
                      *    initialization
                      *    pipe creation <<component 3>>
                      *    filter execution <<component 2>>
                      *    finalization
                      */
```

4.1.5 Summary

After applying operators and developing transformation rules in this manner, we obtain a grammar for the dataflow language. From this, ASCENT can generate an application-specific environment for the dataflow language. This allows us to construct programs such as the one illustrated in Figure 4.2. The resulting environment also contains a program transformer which generated the code for the split filter in Figure 4.9.

PF is a little language, containing 22 terminals and 37 non-terminals. The grammar for C on the other hand, has 32 terminals and 106 non-terminals. In the case study, approximately half of the operators were used to narrow C into C_{pf}, a subset of C used for the initial PF environment. As the PF grammar is broadened to include type definitions and a richer action language, then C_{pf} will become more like C and the potential for sharing between the base and special-purpose environment descriptions will increase, and thus the amount of narrowing will decrease. The narrowing and renaming from C_{pf} account for a third of the operators. These include deleting productions that are not used in the grammar, but never-the-less, are used to define the execution semantics. Restructuring accounts for 15 percent of the operators. Narrowing/renaming and restructuring is where we get the most leverage from the base environment. In our case study, this resulted from the similarity of the PF action language and filter descriptions to similar C structures. Major redefinition is outside of the scope of ASCENT. In the case study, this accounts for 2 percent of the operators. It is used to define two passes over the PF program and to define execution semantics for the connectivity of pipes.

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
#define bool int

/* define pipe data structure */
#define NUM_CONS 4
int fildes[NUM_CONS][2];

     ...     /* define filter routines */

main(argc,argv) int argc; char **argv;
{
int __i;
/* create pipes */
#define split_INP_in 0

pipe(fildes[0]);
#define upper_INP_in fildes[0][0]
#define split_OUTP_left fildes[0][1]

pipe(fildes[1]);
#define lower_INP_in fildes[1][0]
#define split_OUTP_right fildes[1][1]

pipe(fildes[2]);
#define merge_INP_left fildes[2][0]
#define upper_OUTP_out fildes[2][1]

pipe(fildes[3]);
#define merge_INP_right fildes[3][0]
#define lower_OUTP_out fildes[3][1]

#define merge_OUTP_out 1

/* execute filters */
if (0==fork())
   split(split_INP_in,split_OUTP_left,split_OUTP_right);
...

for (__i=0;__i<NUM_CONS;__i++){
  close(fildes[__i][0]); close(fildes[__i][1]);
}
close(0); close(1);
while(-1!=wait(0)){}
}
```

Figure 4.8: Unix Pipes Compilation

```
void split(in, left, right){

   char __buf;
   int __i;
   bool l;

   in = dup(in);
   left = dup(left);
   right = dup(right);

   for (__i=0;__i<NUM_CONS;__i++) {
      close(fildes[__i][0]);
      close(fildes[__i][1]);
   }
   close(0); close(1);

   {
      l = TRUE;
   }

   while(1){
      if (l) {
         __buf = (char)((char) READ(in));
         write(left,&__buf,1);
      } else {
         __buf = (char)((char) READ(in));
         write(right,&__buf,1);
      }
      l = !(l);
   }
}
```

Figure 4.9: Code for Split

ARL code:

```
visit stmtnode son of thenode.arl%default.arl%statements  do
      if HasCommand (stmtnode) then return true
      end if
end visit
```

C **code:**

```
Treenode visit;
visit = _Acc_class (_Acc_class (thenode, _Class_value("arl", "default"), 1),
                    _Class_value("arl", "statements"), 1);
if (_arity (visit) > 0) {
   for (visit = _Acc_first(visit); visit != nil; visit = _Acc_right(visit)) {
      if ((stmtnode = _TrueNode(visit)) == nil)
         break;
      if (HasCommand(stmtnode))
          return true;
   }
}
```

Figure 4.10: Comparison of ARL code and C counterpart

4.2 ARL

The second case study was a retrospective examination of ARL, a special-purpose language for manipulating abstract syntax trees [AKE84]. It is used to write semantic routines for Gandalf programming environments [HN86], and has been in daily use for many years.

As with most application-specific notations, ARL evolved out of the use of a general-purpose programming language, in this case C. Initially abstract syntax trees were processed using C routines. Over time it became clear that C was being used in stylized ways, and that programs could be simplified through a specialized notation. Figure 4.10 illustrates this by comparing a fragment of ARL to its C counterpart. (The code traverses a list of statements and checks if any statement contains a command). This size of code expansion in going from ARL to C is typical.

4.2.1 Transformation Strategy

ARL provides three kinds of special-purpose facilities: (a) a rich collection of primitives for manipulating abstract syntax trees; (b) control structures that make it easy to traverse and search trees (as illustrated in Figure 4.10); and (c) stylized procedure call mechanisms that allow routines in the language to act as event-driven daemons. Because the language incorporates numerous special constructs for manipulating trees, ARL is quite a big language

Category	Level	Support
Easy (16)	1	Automation
Function (3)		
Visit (2)	2	Table Editing
Exception Handling (21)		
Formatting (5)	3	Auxiliary Processing
Cond, Case and Daemon (3)		

Figure 4.11: Classification of ARL Statements and Computation Units

with 177 terminal productions and 127 non-terminal productions. By comparison, the grammar for C has 32 terminals and 106 non-terminals.

Despite this discrepancy in size, we discovered that most of the ARL language could be easily obtained by applying ASCENT operators to C. First, a substantial portion of C is embedded directly in ARL, either identically or with minor renaming. This includes most of C's terminal productions and its expression sublanguage. Of course, many productions in C are not used in ARL. On the other hand, many ARL terminal and non-terminal productions have obvious corresponding C productions, so transformations between them are straightforward using ASCENT narrowing and renaming operators.

Second, we found that most of the remaining special-purpose productions fell into natural categories that could be defined as similar structural operations on C productions. This meant that the same transformation rule could be reused repeatedly with only minor changes. For example, among 46 ARL statement productions, 21 tree manipulation statement productions belong to the same category. Five other statement productions (associated with tree formatting expressions) form another category. As another example, among four basic ARL computation units, three of them (procedure, function and extended command) are natural specializations of the C PROCDEF production. As a whole, those 46 statements (42 non-terminals and four terminals) and four computation units (all non-terminals) were divided into six categories, as was shown in the Figure 4.11. According to the differences in transformation difficulty, the categories were classified into three levels, from easy (Level 1) to hard (Level 3). The corresponding support from ASCENT for the different levels is also shown.

It is interesting to note that all of the other 81 non-terminal productions and all of the other 173 terminal productions belong to Level 1. As the result, almost 90 percent of the total productions belonged to Level 1; about 8 percent belonged to Level 2; only about 2-3 percent belonged to Level 3. We will discuss the support ASCENT provides for transforming productions in each category in the following sections.

```
Boolean function EmptyRetry (node oldnode, newmeta);
  if oldnode.first~opr = _Old_Operator("c","Empty_Retry")
        then return true
        else return false
  end if
```

Figure 4.12: An example of specially-written function

4.2.2 Grammar Operators and Automated Transformation

ARL productions in Level 1 (belonging to the Easy and Function categories) were obtained by using a small number of transformation operators on the appropriate base language (C) productions. At the same time, the transformation rules for mapping the ARL programs back into the C base language were automatically produced and saved. The difference between the Easy and Function categories is in the amount of effort needed in the transformation.

The Easy Category. There are 16 ARL statements belonging to this category. They are: assignment, block, comment, debug, fail, hold code, if, procedure call, two special forms of procedure call: monadic and dyadic, retry, return, skip, succeed, until and while.

The transformation for assignment is typical for statements in this category. First, we *copy* the C statement production and *rename* it as assign, then *lift* the statement as assig and *rename* (actually replace) the right hand side properly. Almost all the others including above mentioned 81 non-terminals and 173 terminals can be treated in similar way or even simpler, (*i.e.*, reusing the production in C directly).

The transformation for retry is an exception in this category. As with several cases in the other categories, here we had to use a specially-written function (see Figure 4.12) to test if its retry part is empty in order to produce the right transformation. However, these functions were typically simple (less than 5-line) boolean functions that determined the context of the transformation.

Perhaps contrary to intuition, it is not always easier to transform terminals of the grammar than non-terminals. Among four ARL terminals in this category: comment, fail, skip and succeed, only the comment production in C is reused in ARL. Dealing with fail and succeed productions is no less complicated than assignment. Nonetheless, their transformation table is short and easy to understand. In the fail example, the transformation defines the semantics of the ARL fail statement as "return false" in C.

The Function Category. As mentioned before, there are three productions belonging to this category: function, procedure, and extended command. Figure 4.13 shows the function

```
C   production:                          ARL production:

PROCDEF = otype procdecl                 FUNCTION = type fun_name formal_part
          lparam body                                comment statement

     Transformation:

               Base:  PROCDEF
               Component 1:   Transform new component 1
               Component 2:   Transform new component 2
               Component 3:   Transform new component 3
               Component 4:   DECLCOMPOUND
                     Component 1:   DECLS
                           List: Skip

                     Component 2:   STATS
                           Component 1: if empty_comment() then
                                          STATEMENT
                                          component 1: SKIP
                                        else
                                          if true Transform new component 4
                           Component 2: Transform new component 5
```

Figure 4.13: Transformation of FUNCTION

production in ARL, the counterpart procedure definition production in C and the corresponding transformation table relating them. The transformation table is automatically produced when the grammar operators are used on the C production to transform it into an ARL function production as follows: *copy* and *rename* procdef as function; *lift* body; *delete* ldecl; *lift* lstat and *rename* (using replace) the right hand side. As with the case of retry, it is necessary here to use a special routine "empty_comment" and properly edit the related part. The procedure and extended command productions are treated in the same way.

4.2.3 Transformation Table Editing

Although they are somewhat more complex, the ARL productions of Level 2 are easily obtained using transformation operators to modify base language productions and introduce new productions. However, in these cases it is typically no longer possible to obtain the related transformations automatically, primarily because of their complicated semantics; some editing of the transformation table is required.

The Visit Category. There are two statements in this category: visit and for. We discuss the visit statement here to illustrate what was involved in one of the more complex

```
DECLCOMPOUND ::= decls stats          VISIT ::= variable node_expr statements

        Base:DECLCOMPOUND
        component 1: /* Treenode visit */
        component 2: STATS
            component 1:STATEMENT
                component 1: ASSIG
                    component 1: IDENT
                        value: "visit"
                    component 2: Transform new comp 2
            component 2: IF
                component 1: /* (_arity(visit) > 0) */
                component 2: FOR
                    component 1: /* visit = _Acc_first(visit) */
                    component 2: /* visit != nil */
                    component 3: /* visit = _Acc_right(visit) */
                    component 4: COMPOUND2
                        component 1: DECLS
                                List: Skip
                        component 2: STATS
                            component 1: IF
                                component 1: EQL
                                    component 1: ASSIG
                                        component 1: Transform new comp 1
                                        component 2: /* _TrueNode(visit) */
                                    component 2: /* nil */
                                component 2: /* break */
                            component 2: DECLCOMPOUND
                                component 1: DECLS
                                        List: Skip
                                component 2: Transform new comp 3
```

Figure 4.14: Transformation Table for “visit” Statement

```
VISIT ::= decls stats

        Base:DECLCOMPOUND
        component 1: Transform new comp 1
        component 2: Transform new comp 2
```

Figure 4.15: First state of Transformation Table for "visit" Statement

grammar transformations. The for statement is handled similarly. The visit construct is a high-level iterator for trees: it traverses a subtree, executing a block of code for each node of that tree. The corresponding code in C is, of course, much more complicated. (See Figure 4.10.)

Figure 4.14 shows the resulting transformation table entry for the "visit" statement; it defines the rules for transforming a "visit" production in an ARL program into its C counterpart. In this case, a large part of the transformation table is devoted to producing fixed C code that will be the same in each transformation. In the figure we represent such concrete syntax as comments. At present, users are required to define this C code in the form of productions created by the transformation table. (We return to this issue in Chapter 6.) For example, the transformation table form for the code of the declaration of "Treenode" (in the first comment) appears in Figure 4.17.

We are able to use grammar grammar operators for most of the transformation. First, we *copied* the declcompound production and *renamed* it to visit—in ASCENT the transformation table shown in Figure 4.15 is automatically produced. We changed the first component, decls, into a *fixed arity* component of one declaration. Then we *lifted* this declaration by replacing it with its children to reveal that a declaration is made up of a type and a list of variables (that may have an initial value)—the transformation table is automatically updated as shown in Figure 4.16. From this state, we *lifted* the type component and chose to make it a user defined type with the value "Treenode." We changed the inidecls component into a *fixed arity* component of one variable declaration and *lifted* each component of the variable declaration to reveal first the variable, which we gave the value "visit", and second the initialization value which we choose to be EMPTY. What we end up with is shown in Figure 4.17. In a similar manner, we *lifted* stats and applied similar operators to the other components to obtain the final transformation table in Figure 4.14.

The purpose of applying operators to the grammar here is somewhat different from the case in Level 1. It is mainly as a simulation of direct editing the table, for our experience shows that the latter is more tedious and error-prone. We are working on devising a tool to facilitate the editing which will be discussed in more detail in Chapter 6.

The Exception Handling Category. The most significant effort occurred in the places where the execution semantics of ARL do not map easily into that of C. For example, ARL provides exception handling in its tree manipulation statements, a control construct that

```
VISIT ::= type inidecls stats

        Base:DECLCOMPOUND
        component 1: DECLS
            component 1: OBJDEF
                component 1:  Transform new comp 1
                component 2:  Transform new comp 2
        component 2: Transform new comp 3
```

Figure 4.16: Second state of Transformation Table for "visit" Statement

```
VISIT ::= stats

        Base:DECLCOMPOUND
        component 1: DECLS                /* Treenode visit */
            component 1: OBJDEF
                component 1:  USERTYPEDEF
                    value: "Treenode"
                component 2:  INIDECLS
                    component 1: INIDECL
                        component 1: IDENT
                            value: "visit"
                        component 2: EMPTY
        component 2: Transform new comp 1
```

Figure 4.17: Third state of Transformation Table for "visit" Statement

```
ARL :                                                    C :

CHANGE          = variable string_expr exception_part    DECLCOMPOUND = decls stats

EXCEPTION_PART = exception_list default                  DECLS        = <declaration>

EXCEPTION_LIST = <exception_elem>                        STATS        = <stat>

EXCEPTION_ELEM = exception_name statements
```

Figure 4.18: Productions related to CHANGE

has no obvious counterpart in C. While we were able to carry out the transformations directly by manipulating the transformation tables—that is, without having to escape to an auxiliary routine—the table entry was complicated since it had to include specific C code to deal with exceptions. It is quite similar to the case of visit category. Happily, the same pattern in transformation table for exception handling can be reused for each of 21 statement transformations.

Figure 4.18 shows the change production, which has an exception part, and Figure 4.19 shows the corresponding transformation. It is a rather complicated transformation An example of the change construct in ARL and its C counterpart is shown in Figure 4.20.

4.2.4 Auxiliary Processing

There are some cases in ARL in which ASCENT cannot provide Level 1 or Level 2 support. The following two categories in Level 3 are examples of where auxiliary processing is needed.

The Formatting Category. This category groups five statements (append, assert, format, pause and stop) together. Each statement has attached to it a format expression which is a kind of abstraction from the C sprintf statement. Figure 4.21 shows an example of an ARL stop statement and its C counterpart. The stop statement will terminate the execution of an ARL program and display the formatted message attached to it. In the example it would be "HeadTail," a concatenation of the strings "Head" and "Tail." Figure 4.22 shows the transformation table for the stop statement. The other statements in this category have similar transformation tables.

It is helpful to take the C counterpart of an ARL construct as a pattern for the construct to be mapped to. The patterns usually are parameterized. In the stop example the number 2 should be replaced by the length of the list which follows the keyword with, the list of assignment statements also depends on the list after with. The design of the pattern is a very important part of our transformational approach, for it is the semantic implementation

```
Structure:  CHANGE
      Base:  DECLCOMPOUND
      Component 1:   DECLS
            Component 1:   OBJDEF
                  Component 1:   USETYPEDEF Value: "Integer"
                  Component 2:   LINIDECL
                        Component 1:   INIDECL
                              Component 1:   IDENT   Value: "_ntry"
                              Component 2:   INITIALIZATION
                                    Component 1:   INTCONST  Value: "0"
            Component 2:   OBJDEF
                  Component 1:   USETYPEDEF Value: "Boolean"
                  Component 2:   LINIDECL
                        Component 1:   INIDECL
                              Component 1:   IDENT  Value: "_retry"
                              Component 2:   INITIALIZATION
                                    Component 1:   IDENT  Value: "true"
      Component 2:   STATS
            Component 1:   WHILE
                  Component 1:   IDENT  Value: "_retry"
                  Component 2:   COMPOUND2
                        Component 1:   DECLS  List:   Skip
                        Component 2:   STATS
                              Component 1:   STATEMENT
                                    Component 1:   ASSIG
                                          Component 1:   IDENT Value: "_retry"
                                          Component 2:   IDENT Value: "false"

                              Component 2:   STATEMENT
                                 Component 1: PROCCALL
                                    Component 1: FIDENT Value: "_Change"
                                       Component 2: LEXP
                                         Component 1: Transform new component 1
                                         Component 2: Transform new component 2
                                         Component 3: ADDRESS
                                            Component 1: IDENT Value: "_result"
                                         Component 4: IDENT Value: "_undoable"
                                         Component 5: IDENT Value: "true"

                              Component 3: STATEMENT
                                 Component 1: AINC
                                    Component 1:   IDENT Value: "_ntry"
                              Component 4: SWITCH
                                 Component 1: IDENT Value: "_result"
                                 Component 2: CASELIST
                                   List: Transform new component 3, component 1, list
                                 Component 3: Transform new component 3, component 2
                                 Component 4: EMP_EMPTY
                  Component 3: EMP_EMPTY
```

Figure 4.19: Transformation for CHANGE

```
ARL code:

    change cn with "funny string"
      when ProtectionViolation =>
           retry 2
      default
           return
      end exception

C code:

    { Integer _ntry = 0;      /* int _ntry = 0;  */
      Boolean _retry = true;  /* int _retry = 1; */
      while (_retry){
        _retry = false;       /* _retry = 0;     */
        _change(cn,"funny string",&_result,...);
      _ntry++;
      switch(_result){
          case ProtectionViolation:
               _retry = (_ntry <= 2);
               break;
          default:
               return;
               break;
      }
     }
    }
```

Figure 4.20: ARL and C versions of a CHANGE instance.

```
ARL Stop Statement Example:

            stop using "%s%s" with "Head", "Tail";

C Counterpart:

            { char * _args[2];
              int _i = -1;

              _args[++_i] = (char *) "Head";
              _args[++_i] = (char *) "Tail";
              _stop(2,"%s%s",_args);
            }
```

Figure 4.21: "Stop" Statement Example and its C Counterpart

of a higher level construct in the base language. A pattern may be very simple or rather complicated depending on the semantics of the particular construct.

It is easy to notice from the transformation table that most of the table is similar to that of previous Level 2 categories, except for the function ArgsStates. This function is needed to map a list of ARL constructs into a special kind of list in C. In this case we must map a list of expressions into a list of assignment statements, where the expression being mapped is part of the assignment. At present, ASCENT cannot express this kind of transformation in the table without using a hand-crafted tree function. This is the reason for defining Formatting as a separate category. The routine numofparas is only a one line function returning the length of the list.

The Cond, Case, and Daemon Category. This category includes ARL cond and case statements the daemon computational unit. Despite their different syntax and semantics, they have a common characteristic in transformation—mapping a list in ARL into a non-list (as an if-then-else construct in C). As with the case in the Formatting category, ASCENT at present has to rely on a specially written routine to support this kind of transformation. The mapping mentioned above is the only transformation needed for the cond statement. The case and daemon constructs need transformations that are similar to those for Level 1 and 2 productions.

4.2.5 Summary

Summarizing our experience in the ARL case study, of the more than 300 terminal and non-terminal productions in ARL, nearly 90 percent of them could be easily transformed from the base language C with an identity transformation, simple grammar operators, or minor editing of the transformation table. For about 8 percent of the productions, we had to edit the transformation table quite heavily. (The treatment of exception handling, mentioned

```
DECLCOMPOUND ::= decls stats              STOP        ::= format_expr
                                          FORMAT_EXPR ::= string_expr exprs
                                          EXPRS       ::= <expr>

          Base:  DECLCOMPOUND

          Component 1:   DECLS
             Component 1:   OBJDEF
                Component 1:   CHAR
                Component 2:   LINIDECL
                   Component 1:   INIDECL
                      Component 1:   PTRDECL
                         Component 1:   ARRDECL
                            Component 1:   IDENT  Value: "_args"
                            Component 2:   INTCONST  Value: Defined by numofparas()
                      Component 2:   EMPTY

             Component 2:   OBJDEF
                Component 1:   USETYPEDEF  Value: "Integer"
                Component 2:   LINIDECL
                   Component 1:   INIDECL
                      Component 1:   IDENT  Value: "_i"
                      Component 2:   INITIALIZATION
                         Component 1:   NEGATE
                            Component 1:   INTCONST  Value: "1"

          Component 2:   STATS
             Component 1:   COMPOUND
                   List:   Replace with tree defined by ArgsStates()
             Component 2:   STATEMENT
                   Component 1:   PROCCALL
                      Component 1:   FIDENT  Value: "_stop"
                      Component 2:   LEXP
                         Component 1:   INTCONST  Value: Defined by numofparas()
                         Component 2:   Transform new component 1, component 1
                         Component 3:   IDENT  Value: "_args"
```

Figure 4.22: Transformation Table for “Stop” Statement

above, is an example.) On the other hand, as we pointed out earlier, the fact that 21 tree manipulation statements belong to the same category greatly reduced the work needed, because we were able to reuse the same pattern of transformation by copying the entry for one of the statements. In the worst cases, it was necessary to escape to auxiliary routines to accomplish the transformation. This accounted for only 3 percent of the productions. Even here we can take advantage of the same pattern of the formating category.

4.3 Implicit Invocation for Ada

In the third case study we added implicit invocation mechanism to Ada [GS93].

Implicit invocation is an increasingly important technique for integrating systems; instead of invoking a procedure directly, a module can announce (or broadcast) one or more *events* [Rei90]. Other modules in the system can *register* interest in an event by associating a procedure with the event. When the event is announced the system run time mechanism invokes all of the procedures that have been registered for the event. Thus an event announcement "implicitly" causes the invocation of procedures in other modules.

An implicit invocation mechanism consists of three components: event declarations, event announcements and event-procedure bindings. Our extension to Ada with implicit invocation provides a straightforward implementation of these components. In particular, Ada package interface declarations are extended to permit event declarations. Ada statements are extended to include an "announce event" statement. For declaring event procedure bindings, we added an event registration program unit. Figure 4.23 shows an example of event declaration, binding, and announcement.

4.3.1 Ada Implementation

We created the implicit invocation environment with ASCENT by extending an existing Ada environment. Programs written using implicit invocation are automatically transformed into legal Ada, including an interface and body for a package called Event_Manager, by a transformer that is the part of the new environment.

Figure 4.24 illustrates how the events declared in Figure 4.23 are represented by the generated event manager package. As illustrated, the list of events is represented as an Ada enumerated type, together with a variant record, which specifies the parameters for each event. In addition, the generated specification contains the interface to the Announce_Event procedure, which allows components to announce events.

Event dispatching is handled by the Announce_Event procedure. This procedure is declared and implemented in a system-generated package, called Event_Manager. As illustrated in Figure 4.25, Announce_Event is structured as a case statement, with one case for every declared event. Dispatching of events is handled by the event system transparently to the announcers of events: modules announce events as part of their normal code by "calling" Announce_Event (Figure 4.26).

```
--example of event declarations

package package_1 is
     declare event_1 (x : integer; y : package_n.my_type);
     declare event_2;
end package_1;

package package_2 is
     declare event_3 (a,b : integer);
end package_2;

--This is an example of event registration, or event procedure binding.

for package_1
    when event_3 => procedure_1 (b);
for package_2
    when event_2 => procedure_4;
    when event_1 => procedure_2 (x);
for package_3
    when event_2 => procedure_3;
    when event_1 => procedure_4 (y);

--example of event announcement

package body package_1 is
begin
    Announce_Event (event_1, x, y);
end package_1;
```

Figure 4.23: Event Specification Language Example

```
with package_n;
package event_manager is
   type event is (event_1, event_2, event_3);
   type argument (the_event : event) is
      record
         case the_event is
            when event_1 =>
               event_1_x : integer;
               event_1_y : package_n.my_type;
            when event_2 =>
               null;
            when event_3 =>
               event_3_a,event_3_b : integer;
            when others =>
               null;
         end case;
      end record;
   procedure announce_event (the_data : in argument);
end event_manager;
```

Figure 4.24: Generated Specification for Event_Manager

4.3.2 Transformation Challenges

Classification of Transformations for Event Management. As with ARL, we classified the transformations needed for the extension of Ada into different categories (see Figure 4.27). As it turns out, the syntactic extensions are such that only three high level changes need to be made to Ada. First is the ability to call Announce_Event. This can be done by adding a new statement type to Ada and as a specialized form of procedure call. The other two changes involve adding the ability to declare events and the ability to register events with procedures. These latter two turned out to be quite tricky.

Event Declaration As illustrated earlier (Figure 4.23), in the new environment events are declared as part of package declarations. Thus event declarations are distributed throughout the program. In the Ada implementation, however, all event declarations are declared in a single package (Event_Manager – see Figure 4.24). Thus it is necessary to transform the distributed information into a central location. Clearly, such a non-local transformation cannot be inferred from local syntactic modifications alone. To handle it we had to escape to an auxiliary processing routine, which walked the tree, collecting all declarations and writing them to a central location. These could then be transformed by the normal operations of ASCENT.

```
with package_1;
with package_2;
with package_3;

package body event_manager is
procedure announce_event (the_data : in argument) is
begin
   case the_data.the_event is
      when event_3 =>
         package_1.method_1 (the_data.event_3_b);
      when event_2 =>
         package_2.method_4;
         package_3.method_3;
      when event_1 =>
         package_2.method_2 (the_data.event_1_x);
         package_3.method_4 (the_data.event_1_y);
      when others =>
         null;
   end case;
end announce_event;
end event_manager;
```

Figure 4.25: Generated Body for Event_Manager

```
package body package_1 is
begin
    announce_event (argument'(event_1,x,y));
end package_1;
```

Figure 4.26: Event Announcement

Category	Level	Support
Announce_Event	1	Automation
Event_Registration	3	Auxiliary Processing
Event_Declaration	3	Auxiliary Processing

Figure 4.27: Classification of Transformations for Event Management

Event Registration Events are bound to procedures as illustrated in Figure 4.23. These bindings result in an Ada routine, Announce_Event, that calls the right procedures when an event is announced. To produce this procedure event-procedure bindings must be transformed into a case statement. The main difficulty is that the procedures bound to a given event must be collected into a single location before transforming them into a case clause of Announce_Event. This was similar to the above problem and yielded to a similar solution.

4.3.3 Summary

This case study was unlike the earlier two case studies. It involved relatively few changes to the grammar itself. One of these changes was trivial, and the other two could not be handled without using external processing to localize distributed information. This case study suggests that there would be significant advantages to pairing ASCENT-like transformations with a declarative notation for localizing information – such as attribute grammars.

Chapter 5

Implementation

Our implementation of ASCENT is based on the implementation of TransformGen. (Recall that TransformGen is a system for updating existing program databases when the grammar for those programs is changed—Section 3.2.) In particular we were able to leverage TransformGen in two significant ways. First, we used TransformGen to prototype ASCENT. This helped to improve our understanding of the limitations of TransformGen and to determine what mechanisms ASCENT would need beyond those of TransformGen. Second, a large amount of the code of ASCENT was taken from TransformGen.

To prototype ASCENT we simulated how it would work on several case studies by using the low-level operators and rule tables of TransformGen to create a transformer. As discussed earlier (Section 3.2), the transformation rules of TransformGen work in the opposite direction from those of ASCENT. Therefore, we prototyped the rule table construction of ASCENT by applying operators to the application-specific environment to obtain the base environment. This gave us the correct rule tables. Although, in the actual system, we would apply operators in the other direction—from the base environment to the application-specific—this exercise provided insight into the kinds of operators that are necessary for ASCENT, driving the creation of higher-level operators and rules.

To implement ASCENT, we modified the existing TransformGen system. While many of the operators in ASCENT are different from those in TransformGen, it was possible to reuse a great deal of the framework. In particular, we could easily adapt the user interface to grammars and to the transformation rule tables. More significantly we were able to adapt the consistency and completeness checks of TransformGen to guarantee that ASCENT transformations produce legal trees (with respect to the base environment).

There were also a number of significant differences between the implementation of TransformGen that of ASCENT. Unlike TransformGen, which replaces an old version of a program with a new version when that program is first read into a new environment, ASCENT must keep both versions of the program since the tree structure generated by the rule tables is the implementation of the application-specific program. Furthermore, we had to adapt the transformation process to record the correspondences between the two tree structures so that tools can easily relate the nodes in the translated program with their source.

Additional modifications to TransformGen required reversing the direction of the transformation rules with respect to the operators, and establishing new correspondences, since the meaning of the operators is quite different. When this was accomplished, we incrementally modified the existing TransformGen rules, and began to introduce additional high-level operators and rules. This required some care in order to preserve the completeness and consistency properties that ASCENT guarantees. ([GKS86] provides a detailed account of these checks.) The initial suite of grammar changing operations in ASCENT Version 1 included: adding and deleting productions to and from the grammar, adding and deleting components to and from a production, and copying and renaming productions. Version 2 added new higher level operators such as lift and lower.

In the remainder of this chapter we consider the implementation in more detail.

5.1 ASCENT as a Language-Oriented Environment

ASCENT is implemented using the Gandalf environment generator [HN86], which provides a great deal of power, both in rapid generation of environments and in controlling the side effects of a user's actions. In particular, a grammar is treated as a language, and the user interface to ASCENT is a language-based editor. Operators applied to a grammar are then special-purpose commands provided by the editor. The rule table is produced as a side effect of manipulating the grammar. This is accomplished using the Gandalf "daemon" mechanism, which allows the environment to associate auxiliary actions with any editor command. This mechanism allows ASCENT to (a) track changes to the grammar, (b) automatically update the transformation tables so that they are always self consistent, and (c) infer the desired contents of the tables.

Similarly, the tables produced by the editing daemons are treated as a language. This allows the user to edit the tables directly using another set of editing commands, in this case the commands that provide rule modification operations.

Additionally, throughout the editing process ASCENT maintains a copy of not only the current grammar, but also the base grammar (using daemons associated with many of its own grammar productions to provide copy-on-write semantics). Although the base grammar is usually invisible to the user, it can be used to fill in default information in transformation rules as well as to check the validity of these rules when they are complete.

The transformation rules themselves are associated with each production in the new grammar version, and may be modified in three ways.

- The user may invoke a grammar modification operation using one of the operators provided by ASCENT. These operations correctly update the grammar and the transformation rules in parallel and can therefore easily take whatever action is appropriate.

- The user may use lower-level language-oriented editing operations to modify a grammar. If the operations change the structure of the grammar, then the daemons will locate the appropriate transformation rules and update them appropriately. Since the daemons are specifically associated with different components of the grammar's

abstract syntax, each editing operation can have different effects depending upon the information it is modifying.

- The user may directly edit a transformation rule without changing the grammar itself. As we have seen, this is useful when the user desires some transformation other than obvious one inferred by the system. (This is almost always a result of a complex and non-obvious goal – the system seldom makes an "incorrect" guess.)

The actual translation of the user program into the base language is performed by a table driven transformation engine. (Although, the external interface is different, this is the same transformation engine used for TransformGen.) The transform engine interprets the rules provided by the editor implementor to create a base language tree from the information contained in the user's program tree. (Although the structure of the resulting tree will typically be similar to that of the source tree, the implementor is free to specify arbitrary constructions if needed.)

The transform engine operates by taking the transform rule associated with the root of the source tree and constructing the node described by the rule. The components of this node are determined by recursively invoking the transformer in the manner specified by the rule. Components may themselves be explicitly constructed or they may by computed by interpreting the rule associated with another implementor specified node. (Typically this is one of the components of the source node.)

5.2 Reusing the TransformGen Implementation

The major benefit of building upon TransformGen was the opportunity to re-use the general framework provided by that implementation. We were able to adopt the transform table structure without any changes at all by simply duplicating the appropriate component in the production description. Likewise, the "unparse schemes" which control the C code written out in the grammar description could be adapted with only minimal modification (*i.e.*, referencing the new component rather than the old, and changing the times at which the tables are written).

The implementation also uses much of the old code for the parallel maintenance of "new" and "old" grammar versions, and for detecting changes that can affect the transform tables. However, since the actions required to update ASCENT transform tables are substantially different than those of TransformGen, parallel code had to be written for all of the various update actions. (Of course, the explicit ASCENT operations such as "lift" and "lower" had to be coded from scratch, although similar operations may be later added to TransformGen.)

It was also necessary to modify some of the existing framework so that it would distinguish between standard and ASCENT transform tables in various operations that depend upon this context. (For example, when a new production is added, the code that finds the production description and creates a template must know whether it should search for the new or the old version of a production.)

The kernel transformation engine was also able to re-use most of the existing code. The interpretation of the transform tables and the transformation process itself are identical regardless of the direction of the transformation. However, some additions were required. A new interface function was added to allow explicit triggering of ASCENT transformations distinct from the implicit triggering of standard transactions. Additional code was added to the transformer itself to maintain a mapping between transformed nodes and their sources (to allow for easier error reporting by the environment developer) and to add a small number of additional options to the transformation rules themselves. These additional options include "tree functions" to define subcomponents and a capability for merging multiple lists into a single list.

The implementation effort took approximately three weeks. The comparable effort to produce the first TransformGen system, building from the Gandalf System [HN86] required over three months. Implementation of a transform engine was also required as part of this effort, and required the addition of a new module with approximately 30 routines. Although we expect that further modification will be necessary to give ASCENT the capabilities we ultimately envision, we have been able to get by with minimal changes to the transform engine. Providing the functionality of ASCENT, Version 2, added another three weeks to the implementation effort and required adding six additional routines and changing approximately 20.

Chapter 6

Assessment of ASCENT

In our development of ASCENT we performed the dataflow and ARL case studies three times. First, we used the existing implementation of TransformGen to prototype ASCENT. From this effort, we determined which mechanisms were needed to make the approach viable. These included operators for manipulating grammars, new restructuring operators not found in TransformGen, and transformation patterns to express context-sensitive restructuring and the corresponding transformation rules.

After implementing our initial version of ASCENT, we returned to the case studies to evaluate how well it performed. While we were happy with the support it was giving, we observed an opportunity to do even better by adding more powerful grammar-changing operators and operators for modifying transformation rules. These included the restructuring operators, "lift", "lower", "make-fixed-arity", "duplicate-var-arity", and a more general transformation rule for constructing variable arity productions. We added these to produce ASCENT, Version 2, and worked through the case studies a third time. In addition, we performed the third case study, in which we added a implicit invocation mechanism to Ada.

Our experience has been that adding more powerful operators makes a significant difference in the ease with which ASCENT can be used to develop application-specific environments. Figure 6.1 illustrates this effect. The effort required to develop a new grammar and associated transformer can be divided into three parts: performing operations on the grammar, modifying the rules to handle situations not automatically inferable from the grammar operations, and adding auxiliary processing to handle transformations outside the scope of the transformation rules. Without *any* grammar operators (Figure 6.1 A) the implementor is forced to code a transformation by hand for the entire language. (This is effectively the effort of producing a preprocessor by hand.) With a simple set of grammar operators (such as those of the initial version of ASCENT) some of the transformations can be automated. However, typically there is still considerable work to be done to manually augment the transformation rules, and, moreover, a certain amount of auxiliary processing remains (Figure 6.1 B). As more powerful operators are introduced (such as those of the ASCENT Version 2), less effort is required to produce transformation rules and simultaneously less in writing auxiliary processing routines (Figure 6.1 C). Since applying grammar

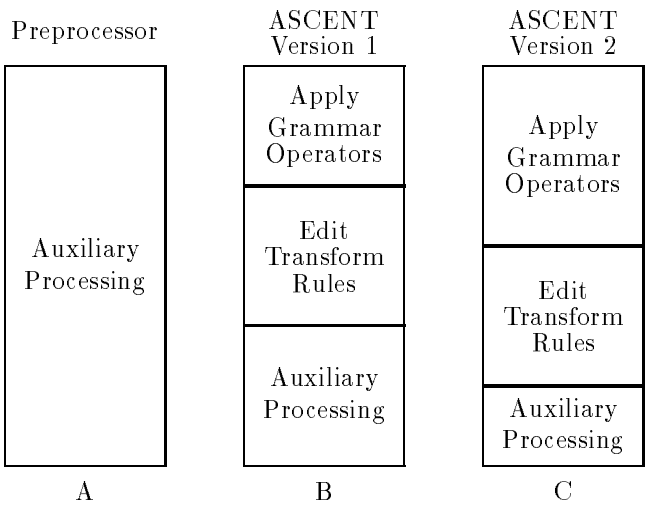

Figure 6.1: Distribution of Effort

operators is much easier (and less error-prone) than developing auxiliary routines, the total effort is reduced. For example, in the two case studies 80-85 percent of the transformation rules can be derived directly from grammar operators, 10-15 percent required editing of the transformation rules, and only 2-5 percent needed to resort to auxiliary processing.

Overall, we have found that ASCENT works well in cases where there is moderate restructuring but less well for situations that require a major redefinition of the base environment. The case studies illustrate this difference: we achieved considerable leverage where moderate restructuring is required (for example, extended commands in ARL, and filters in PF).

The hard cases arise when the underlying execution model is different. Here it is helpful to distinguish between structure-oriented and execution-oriented transformation. Grammar operators are primarily oriented towards structural changes. However, when the computational model of application-specific programs differs from that of the base environment, we must also provide transformation rules that bridge those gaps. In general, these cannot be inferred from structural operators alone, but must be tailored by the implementor. This happened in the PF example when we needed to provide execution semantics for pipes. Similarly, the dispatch code of the Ada event extensions must be coded from scratch. In the ARL example, we obtained more leverage because it is closer to C than PF, although non-C constructs, such as exception handling, still remained difficult.

There are two reasons for this. The first source of difficulty is that bridging the semantic gap typically requires introduction of specific code to handle program execution. For the PF example, this included the code to create and connect pipes, fork processes, etc. With ARL, for example, much of the transformation table entry for the "visit" production (Figure 4.14) simply builds a tree that contains specific C code. In the Ada extensions it included the code

```
VISIT = variable node_expr statements

   Base:
      {
         Treenode visit;

         visit = <<Transform comp 2>>;
         if (_arity(visit) > 0)
            for (visit = _Acc_first(visit); visit != nil; visit = _Acc_right(visit)) {
               if (<<Transform comp 1>> = _TrueNode(visit) == nil)
                  break;
               {
                  <<Transform comp 3>>
               }
            }
      }
```

Figure 6.2: Transformation Template for the "visit" Production

for the dispatcher. The reason that this is difficult for ASCENT is that the transformation tables do not currently lend themselves to including specific pieces of code.

The second source of difficulty is that sometimes it is necessary to centralize information distributed throughout one program. In the Ada example, this was needed to collect distributed event declarations into a single implementation module. Such global restructuring is beyond the automatic capabilities of ASCENT.

To address the first problem we have begun work on developing a way to generate transformation rules "by example" [KW87, Wil92]. Using this approach, instead of directly editing the transformation table, the implementor first creates a code template in the base language. This includes all of the concrete, default syntax that is to be provided around the parts that are otherwise transformed. The parts of the template that remain to be filled in are left as placeholders – or parameters – of the template. The template is used in a transformation rule by supplying tree expressions for each of the placeholders (recursively defined by the tables). ASCENT then automatically creates the expanded transformation using the template. Figure 6.2 illustrates the form of a template for the "visit" table entry. Comparison with Figure 4.14, makes it clear why this represents is a significant improvement.

The second problem – localization of information – could be handled by adding new transformation operators and providing new transformation algorithms. For example, an initial pass could collect the data and then a second pass could transform it. However, we believe that a much more promising approach is to make use of facilities for manipulating information in language-based editors. In particular attribute grammars or special kinds of daemons could be used to collect distributed information that could then be used by ASCENTin the transformation process.

Our case studies have taught us that despite the need to tailor transformation rules in the few hard cases, ASCENT provides significantly better support than building a transformer by hand. In particular, its support for guiding the construction of transformation rules and for checking their validity is invaluable. Moreover, by comparison to producing a text preprocessor for the new language, the tree-to-tree transformer produced by ASCENT is far superior. In particular, it allows us to reuse syntax-directed tools of an existing base environment. This is further assisted by providing backpointers from nodes in a transformed program tree to those in the source.

Chapter 7

Related Work

The desire to exploit commonalities in families of software systems is pervasive in much of the recent research in software engineering. Consequently this research is closely related to a number of other research efforts. These can be divided into two basic categories.

7.1 Reuse

Research on software reuse shares with this research the goal of exploiting commonalities in software systems. However, most work on software reuse focuses primarily on issues of software *component* reuse. A software component is typically taken to be a software entity at the level of a procedure, module, or abstract data type. The key research issues there are to understand how to characterize, store, retrieve, catalogue, browse, protect, distribute, and adapt individual components [BP89, Fre83, Boo87].

In contrast, our research is more concerned with *design* reuse. An application framework represents the common infrastructure for a family of applications, and thus determines the overall form of an application. As such, framework-based reuse and component-based reuse are complementary. Having chosen an application framework the software engineer could in principle "plug in" various reusable components to build a particular application of the framework. For example, a signal processing application might adopt the dataflow framework illustrated earlier, but then choose the component actions from a library of signal processing routines.

Closer in intent to our use of frameworks has been work on program *schemas* [KRT87, VK85, Wat85] and *frameworks* as reusable object-oriented components [Deu89]. In most of this work, parameterized templates are stored in a library and later instantiated to produce portions of a program. Our work, while motivated by similar issues, differs from research on reusable program schemas and object-oriented components insofar as it attempts to build software commonalities into notations and environments that are specialized to the particular family of applications. The application-specific support can be much more pervasive, and more fully automated in our approach.

7.2 Multi-Paradigm Environments

Several research projects have considered the problem of developing specialized notations for application domains [Mul86, Fre87, IWJ90, Wil92]. Among these the work here is most closely related to that by Neighbors [Nei84] and by Wile [IWJ90, Wil92]. In both of these efforts domain-specific languages are transformed into lower-level (typically executable) notations. There are, however, two important differences between these efforts and ours.

First, in both of these efforts, as well as other research into transformational approaches to reuse [Gol90, Che84, Fea83, L+88], the primary technology exploited is general-purpose program transformation: an arbitrary high-level language is successively transformed into a lower level one suitable for execution. In contrast, our application-specific environments use program transformation in a much more limited way. Because the higher level notation was developed by transforming an existing base level notation, the gap between the two is much smaller. Moreover, because the domain-specific notation was developed through a transformational process itself, the inverse translations become much more easily automated. Thus while a general transformational approach provides more flexibility in choice of domain-specific language, it also incurs the overhead of introducing much more complex machinery for performing the transformations.

Second, both of these efforts, and others in the field of multi-paradigm research [Zav89, Mul86], are concerned with the general problem of combining several programming paradigms into the same environment. This is typically viewed as a problem of interfacing: how does one connect components written in different languages. Approaches to the problem include use of structure sharing [IWJ90, Lam87], abstract module interfaces [Pur90], and transformation to intermediate form [Nei84, Rei87]. In contrast, ASCENT is used to specialize a given base environment and therefore incorporates portions of the base language into the resulting framework. Hence it does not have to solve the more general problem of interfacing arbitrary languages.

Chapter 8

Conclusions

The contributions of this work are twofold. First, we have developed a new technique for producing application-specific environments: a framework is incorporated into a general-purpose environment through a process of structured transformation. We have illustrated that it is both possible and practical to do this by carrying out case studies that develop non-trivial application-specific programming environments from an existing C programming environment.

Second, we have shown that it is possible to automate significant portions of the process. To do this we have extended existing techniques that use a transformational approach, but were developed to solve a different problem (viz., database evolution). While this work was done in the context of structure-oriented environments, the transformations themselves could apply to any system in which programs are represented as abstract syntax trees. In particular, ASCENT's operators provide a comprehensive set of transformations and rules that could be reused in many other transformation-based systems (Appendix A).

This work can be viewed as having successfully applied preprocessing techniques to programming environments. But this is not the whole story. First, our system makes preprocessing tractable by partially automating the production of the translator. Second, the translation process is not simply a text-to-text conversion, but is embedded in the context of a programming environment, supporting links between the two representations of a program. This makes it possible for tools to relate the two representations. This does not address all the problems of dealing with the output of tools on the translated program (for example, the text of compiler errors). But it does allow tools to exploit the direct connection in an interactive fashion. Third, it also allows the translator to handle auxiliary information such as attributes that are not reflected in the textual representation of a program.

In evaluating the overall effectiveness of this approach, we can identify a spectrum of application-specific environments depending on how much they differ in essential ways from a given general-purpose environment. As our case studies have illustrated, significant parts of non-trivial application-specific environments can be handled using the techniques. On the other hand, the more different the underlying execution model, the less applicable our

techniques. In the limiting case—as between Assembler and C—the languages may be so different that no amount of automated assistance will substitute for the need to provide a compiler.

But perhaps more importantly, this point of view indicates that our work can be viewed as establishing a kind of lower bound on what is possible to automate. We can well imagine that new operators and more flexible transformational languages will make it possible to push the frontier of the approach. Moreover, it should also be possible to introduce other kinds of automated support to bridge gaps not handled by automatic rule generation. In particular, we expect that many of the difficult cases that we encountered will be more easily handled by mechanisms described in Chapter 6, whereby transformation rules are obtained by abstracting over existing general-purpose code.

Acknowledgement. We thank David Miller for his invaluable assistance in guiding us through TransformGen for the preliminary case studies and for his insights into implementing the initial ASCENT system. We would also like to thank other members of the Gandalf and Able research groups for their constructive comments on ASCENT and the approach that it embodies.

Bibliography

[AKE84] Vincenzo Ambriola, Gail E. Kaiser, and Robert J. Ellison. An action routine model for ALOE. Technical Report CMU-CS-84-156, Carnegie Mellon University, August 1984.

[Bal85] Robert Balzer. Automated enhancement of knowledge representations. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 203–207, 1985.

[BCD+88] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Boston, MA, November 1988. SIGSOFT '88.

[BGV90] Robert A. Ballance, Susan L. Graham, and Michael L. VanDeVanter. The Pan language-based editing system for integrated development environments. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 77–93, Irvine, CA, 1990. SIGSOFT '90.

[Boo87] Grady Booch. *Software Components with Ada: Structures, Tools, and Subsystems.* Benjamin/Cummings Publishing Company, Inc., 1987.

[BP89] Ted J. Biggerstaff and Alan J. Perlis. *Software Reusability.* ACM Press, 1989.

[Che84] Thomas E. Cheatham, Jr. Reusability through program transformations. *IEEE Transactions on Software Engineering*, SE-10(5):589–594, September 1984.

[Cle88] J. Craig Cleaveland. Building application generators. *IEEE Software*, 5(4):25–33, July 1988.

[Deu89] Peter L. Deutsch. Design reuse and frameworks in the Smalltalk-80 programming system. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability.* ACM Press, 1989.

[DG90] Norman Delisle and David Garlan. Applying formal specification to industrial problems: A specification of an oscilloscope. *IEEE Software*, September 1990.

[DGHKL84] Veronique Donzeau-Gouge, Gerard Huet, Gilles Kahn, and Bernard Lang. Programming environments based on structured editors: The Mentor experience. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments.* McGraw-Hill Book Co., 1984.

[Fea83] Martin S. Feather. Reuse in the context of a transformation based methodology. In *Workshop on Reusability in Programming*, pages 50–58, Stratford, Connecticut, September 1983. ITT Programming.

[Fre83] Peter Freeman. Reusable software engineering: Concepts and research directions. In *Workshop on Reusability in Programming*, pages 2–16, Stratford, Connecticut, September 1983.

[Fre87] Peter Freeman. A conceptual analysis of the Draco approach to constructing software systems. *IEEE Transactions of Software Engineering*, SE-13(7):830–844, July 1987.

[GKS86] David Garlan, Charles W. Krueger, and Barbara J. Staudt. A structural approach to the evolution of structure-oriented environments. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Palo Alto, CA, December 1986.

[GM84] David Garlan and Phillip L. Miller. Gnome: An introductory programming environment based on a family of structure editors. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

[Gol90] Alan Goldberg. Reusing software developments. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 107–119, Irvine, CA, 1990. SIGSOFT '90.

[GS93] David Garlan and Curtis Scott. Adding implicit invocation to traditional programming languages. In *Proceedings of the Fifteenth International Conference on Software Engineering*, Baltimore, MD, May 1993.

[HKN85] Ellis Horowitz, Alfons Kemper, and Balaji Narasimhan. A survey of application generators. *IEEE Software*, 2(1):40–54, January 1985.

[HN86] A. Nico Habermann and David S. Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, SE-12(12):1117–1127, December 1986.

[IWJ90] Edward A. Ipser, Jr., David S. Wile, and Dean Jacobs. A multi-formalism specification environment. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 94–104, Irvine, CA, 1990. SIGSOFT '90.

[KRT87] Shmuel Katz, Charles A. Richter, and Khe-Sing The. Paris: A system for reusing partially interpreted schemas. In *9th International Conference on Software Engineering*, pages 377–385, Monterey, CA., March 1987.

[KW87] Eugene E. Kohlbecker and Mitchell Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich*, pages 77–84. ACM, January 1987.

[L+88] Peter Lee et al. The Ergo Support System: An integrated set of tools for prototyping integrated environments. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Boston, MA, November 1988. SIGSOFT '88.

[Lam87] David A. Lamb. Sharing intermediate representations. *ACM Transactions on Programming Languages and Systems*, 9(3), July 1987.

[LH90] Barbara Staudt Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. In *OOPSLA/ECOOP '90*, Ottawa, Canada, October 1990.

[Mul86] Special issue on multiparadigm languages and environments. IEEE Software, January 1986.

[Nei84] James M. Neighbors. The Draco approach to constructing software reusable components. *IEEE Transactions on Software Engineering*, SE-10(5):564–574, September 1984.

[Pur90] James M. Purtilo. The Polylith software bus. Technical Report CS-TR-2469, Department of Computer Science, University of Maryland, May 1990.

[Rei87] Steven P. Reiss. Working in the Garden environment for conceptual programming. *IEEE Software*, 4(6), November 1987.

[Rei90] Steven P. Reiss. Connecting tools using message passing in the field program development environment. *IEEE Software*, July 1990.

[RT84] Thomas Reps and Tim Teitlebaum. The Synthesizer Generator. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

[RTD83] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3), July 1983.

[SZ90] Andrea H. Skarra and Stanley B. Zdonik. The management of changing types in an object-oriented database. In *OOPSLA/ECOOP '90*, Ottawa, Canada, October 1990.

[VK85] Dennis M. Volpano and Richard B. Kieburtz. Software templates. In *8th International Conference on Software Engineering*, pages 55–60, Imperial College, London, UK, August 1985. IEEE Computer Society Press.

[Wat85] Richard C. Waters. The Programmer's Apprentice: A session with KBEmacs. *IEEE Transactions on Software Engineering*, SE-11(11):1296–1320, November 1985.

[Wil92] David S. Wile. Integrating syntaxes and their associated semantics. Technical report, ISI/USC, 1992.

[Zav89] Pamela Zave. A compositional approach to multiparadigm programming. *IEEE Software*, September 1989.

Appendix A

Operators and their Corresponding Rules

A.1 Narrowing/Renaming

The result of applying narrowing and renaming operators to a grammar is an application-specific grammar that is effectively a subset of the original, (subject to renaming). Trees constructed with the application-specific environment will be valid trees in the base environment. Hence only trivial transformations are needed to handle these changes.

- *Delete production.* Delete a production from the grammar. There is no transformation rule necessary since this production does not appear in the new grammar.
- *Delete choice.* Delete a production from a set of choices on the right hand side of a production. The resulting default transformation rule leaves the structure unchanged since the choices provided are a subset of the original.

```
P ::= { c1 | c2 }              P ::= c1

          Base: P
          Component 1: Transform new component 1
```

- *Rename.* Rename a production to reflect the new application. The resulting default transformation leaves the structure unchanged, but renames the production.

```
P ::= c1 c2                    Q ::= c1 c2

          Base: P
          Component 1: Transform new component 1
          Component 2: Transform new component 2
```

- Copy rename. Unlike rename, copy-rename makes a copy of a production leaving the original production in the grammar. This is useful as a prelude to further specialization.

```
P ::= c1 c2                    P ::= c1 c2
                               Q ::= c1 c2
          Base: P
          Component 1: Transform new component 1
          Component 2: Transform new component 2
```

A.2 Restructuring.

The restructuring operators handle two kinds of situations: (1) when application-specific programs provide the same kinds of information as the base environment, but are structured in a different way (*e.g.*, reversed components); and (2) when a variable portion of the base environment is implicitly defined in the application environment (*e.g.*, the value of a deleted component).

- *Remove component.* Delete a component of a production. The resulting default transformation rule leaves a placeholder for the deleted component, filling in an as-yet-to-be-determined value supplied by the implementor. Components are renumbered appropriately.

```
P ::= c1 c2 c3                 P ::= c1 c3

          Base: P
          Component 1: Transform new component 1
          Component 2: Skip
          Component 3: Transform new component 2
```

- *Exchange components.* Reorder the components of a production. The resulting default transformation rule swaps the corresponding components.

```
P ::= c1 c2 c3                 P ::= c2 c1 c3

          Base: P
          Component 1: Transform new component 2
          Component 2: Transform new component 1
          Component 3: Transform new component 3
```

- *Make fixed arity.* Change a list component to fixed arity. The resulting default transformation rule gathers the fixed number of components into a list.

```
P ::= <c>                          P ::= c ... c

          Base: P
          List: Transform new component 1
                   ...
                Transform new component n
```

- *Duplicate var arity.* Duplicate a variable arity component (effectively breaking the list into smaller components). The resulting default transformation rule appends the duplicated components into a list.

```
P ::= c                            P ::= c c
c = Q
Q ::= <x>

          Base: P
          Component 1: Q
             List: Transform new component 1
                   Transform new component 2
```

- *Lift.* Replace a component by its children. The resulting default transformation rule builds the structure that was lost in replacing the component by its children.

```
P ::= c1 c2 c3                     P ::= c1 d1 d2 c3
c2 = Q
Q ::= d1 d2

          Base: P
          Component 1: Transform new component 1
          Component 2: Q
             Component 1: Transform new component 2
             Component 2: Transform new component 3
          Component 3: Transform new component 4
```

- *Lower.* Replace a group of components by a new production whose children are the group of components. The resulting default transformation rule extracts the appropriate components from the structure that was introduced into the new program and builds the corresponding ("flattened") base program.

```
P ::= c1 c2 c3 c4                  P ::= c1 b c4
                                   b = Q
                                   Q ::= c2 c3
```

```
Base: P
Component 1: Transform new component 1
Component 2: Transform new component 2, component 1
Component 3: Transform new component 2, component 2
Component 4: Transform new component 3
```

- *Make nonterminal.* Change a terminal to a nonterminal. The resulting default transformation rule builds the structure for the terminal and leaves a placeholder for the value to be filled in by the implementor.

```
P ::= lexterm                    P ::= c1 c2

          Base: P
          Value: Skip
```

- *Make terminal.* Change a nonterminal to a terminal. The resulting default transformation rule builds the structure for the nonterminal and leaves a placeholder for each of the components to be filled in by the implementor.

```
P ::= c1 c2                      P ::= lexterm

          Base: P
          Component 1: Skip
          Component 2: Skip
```

A.3 Major Redefinition.

These operators are used to add new productions and components to a grammar. They arise when there is no direct analogue in the base language for the new construct. Since there is no corresponding production in the base grammar the implementor must provide the translation for this new production by directly editing the rule tables.

- *Add production.* Add a production to a grammar.
- *Add component.* Add a component to a production.
- *Add choice.* Add a production to the set of choices on the right hand side of a production.