Beyond LEX and YACC: How to Generate the Whole Compiler

W. M. Waite
Department of Electrical and Computer Engineering
University of Colorado
Boulder, Colorado

Abstract

LEX and YACC (or FLEX and BISON) allow a compiler writer to generate scanners and parsers from simple specifications, but scanning and parsing account for only about 15% of a typical compiler. Tool support is now available for the other 85%. This paper explains how new and existing tools have been combined in a domain-specific programming environment that reduces compiler development effort to about one third of that needed with LEX and YACC alone. Measurements show that the resulting compiler's performance is within a few percent of a carefully hand-crafted version.

1 Introduction

LEX and YACC (or FLEX and BISON) are useful because they allow a designer to describe the scanning and parsing problems to be solved, rather than requiring detailed solutions. Such descriptions, which explicitly list properties that a solution must have, are called *declarative specifications*. A program is an *operational specification*, which gives a recipe that has the desired properties without listing those properties explicitly.

Declarative specifications are easier than operational specifications to develop and maintain, because they describe the properties of the problem rather than the details of the solution. The cost of this advantage, however, is that designers and maintainers must learn more languages in order to use declarative specifications than to use operational specifications. For example, to specify scanners and parsers declaratively, a designer must learn the languages of regular expressions and context-free grammars; only a programming language must be learned to specify them operationally.

A declarative specification must ultimately be turned into code if the problem it describes is to be solved. The mechanism for this translation may be a program, or it may be a human. Having a program to do the translation greatly increases the usefulness of the specification, but it also requires an investment in software. Such investments only make sense when many instances of the given problem, each with different properties, must be solved. Clearly the scanning and parsing problems fit this profile, and hence a number of software tools are available for translating regular expressions and context-free grammars into scanning and parsing code.

If a technique is to be applicable to a large number of problem instances, it is usually incapable of describing any of those instances completely. Thus a mixture of declarative and operational specifications will be required. For example, a context-free grammar can describe a parsing problem, but that parsing problem is only a part of the compilation problem. The declarative context-free grammar is therefore augmented by operational specifications in the form of code fragments to be executed when the parser recognizes certain phrases.

As our understanding of a class of problems improves, we usually are able to recognize more things that are common to solutions of problems in that class. These common aspects of the problem solutions can then be embodied in additional declarative specification techniques for the problem class. The compilation problem has certainly followed this route: Algorithmic conversion of expressions to a machine-oriented form was first described in 1952 [12], context-free grammars appeared in 1956 [2] and were used as a declarative specification in the ALGOL 60 Report [11]. Context-free grammars were initially translated to code by hand using recursive descent, a technique that remains viable today [10].

In Section 2 I will show how the use of declarative specifications can be extended beyond the scanning and parsing subproblems of the compilation problem. Section 3 explains the concept of a domain-specific programming environment that embodies our understanding of the structure of a compiler, and shows how

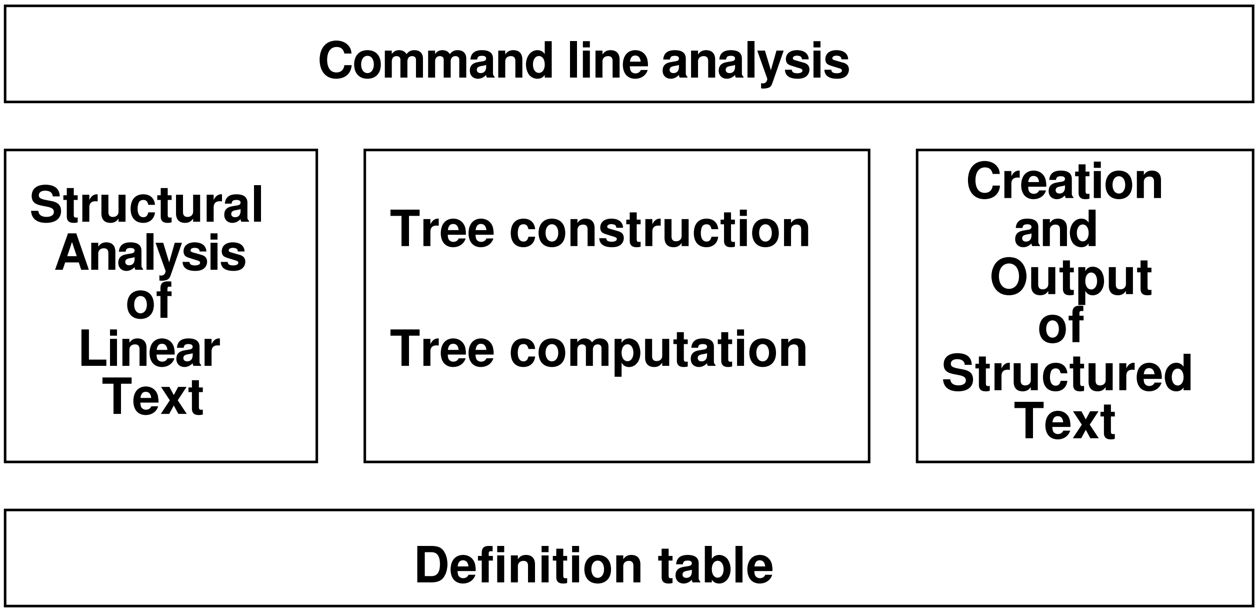

Figure 1: A Typical Compiler

it is used to integrate the tools that translate the declarative specifications. A complete example of a simple compiler is given in Section 4.

2 Describing a Complete Compiler

Figure 1 shows a modular decomposition of a typical compiler. Each block represents a module that could be generated by one or more tools. For example, LEX and YACC can generate the structural analysis module from declarative specifications. All of the modules represented by the other four blocks can also be generated from specifications by existing tools. Ideally, all of these specifications would be declarative; in practice, some operational specifications are required.

The specifications and generation tools for the structural analysis module are well known. In the remainder of this section I will briefly discuss those for the remaining four modules.

2.1 Definition table

The definition table is a data base that the compiler uses to store arbitrary information about entities of interest. Typical entities are user-defined variables, types, and labels. Because many of these entities are represented in the source program by symbols, the definition table is sometimes called a "symbol table" and its role restricted to storing information about such entities. This restriction is a mistake, because it forces the compiler designer to devise other mechanisms for storing information about entities that are not represented by program symbols.

A general definition table module represents each entity by a *definition table key*. It exports a function `NewKey()` that delivers a new, unique definition table key at each invocation, and a distinguished value `NoKey` to represent no entity. The module is capable of storing an arbitrary number of named *properties* for each key. It exports a query operation and an update operation for each property. The query operation takes two arguments, the definition table key for the entity whose property is being queried and a default value of that property to return if the property is not defined for that key [15].

To describe a general definition table module of this kind, all that is required is a list of the names and types of the properties that the module must support. A simple declarative specification of the following form suffices:

```
Size:  int;
Type:  DefTableKey;
Grump:  grumptype; "grumpmod.h"
```

Here `int` is a primitive C type and `DefTableKey` is the type of a definition table key, so that neither requires further specification. A `grumptype`, on the other hand, is exported by the user-defined grump module and therefore the interface for that module (`grumpmod.h`) must be supplied as part of the specification.

The obvious translation of such a specification into a definition table module is performed by the PDL tool.

2.2 Command line analysis

A common form for a command line in Unix is a program name, followed by an arbitrary number of *options*, followed by an arbitrary number of *positional parameters*. Options might be Boolean (the presence or absence of the option is important), or they might be associated with integer or string values; positional parameters are normally strings that represent file names.

The format of a command line can be described by a simple declarative specification that gives names to the various options and positional parameters and specifies the type of each value. For example, some common characteristics of the `cc` command line could be described as follows:

```
Debug "-g"boolean;
OutputFile "-o"string;
InputFiles positionals;
```

A tool called CLP accepts this specification and generates a module exporting variables `Debug`, `OutputFile` and `InputFiles`. The value of `Debug` will be set to 1 if the `-g` option is specified on the command line, and 0 otherwise. If the `-o` option is specified, the string argument will be stored and associated with a definition table key. This key will be the value of `OutputFile`. Otherwise the value of `OutputFile` will be the distinguished value `NoKey`. `InputFiles` will point to a list of definition table keys, one associated with each positional parameter string. This list may be empty.

The remainder of the compiler can access the information from the command line by accessing the variables exported by the module CLP creates. (Note that this module depends on the definition table module.)

2.3 Creation and output of structured text

We create structured text by pasting together text fragments. Each fragment can be created by invoking a routine that constructs the fragment from its component parts. Usually there is a hierarchical relationship among the fragments, with some fragments being constructed from basic data types like integers and strings and others being constructed from previously-constructed fragments.

The format string in a `printf` call is a declarative description of a text fragment. It specifies certain constant parts, and certain places where computed information can be inserted. A similar technique, describing each text fragment as a combination of strings and other fragments, can be used in a declarative specification of the output module for a compiler:

```
While:
  "JMP "$4 "\n"
  $3 ":  "$2
  $4 ":  "$1
```

This specification might describe the assembly language corresponding to a C `while` statement. Constant parts are described by C strings, other fragments by `$` signs. The digit identifies a particular fragment. A tool called PTG will translate this description into a C routine named `PTGWhile` that takes four text fragments as arguments and delivers one text fragment as its result. The first argument to `PTGWhile` would be the translation of the test of the `while` statement, and the second would be the translation of the body. If the test yielded true, the code would transfer control to a generated label supplied as the third argument to `PTGWhile`. Finally, the fourth argument to `PTGWhile` would be a new generated label.

The digits may be omitted in the specification if each component fragment is inserted exactly once, and the order of the arguments is the same as the order of the component fragments. All of the examples in Section 4.2 take that form.

A structured text module generated from a PTG specification exports one function for each named text fragment. Each of these functions returns a data structure called a `PTGNode` that represents the completed fragment, which is the root of a directed acyclic graph of fragments. This `PTGNode` can then be passed to other routines of the generated module for use as components of other fragments, stored in the definition table as properties of entities, or output as text.

2.4 Tree construction and computations

Any tree computation algorithm has three components: a set of computations done at each node, a traversal strategy, and a mechanism for managing intermediate storage. When the algorithm is written by hand, the author must design and implement all three components. It is well known, however, that the traversal strategy and the storage management mechanism can be deduced from the dependences among the computations at the nodes [16].

Attribute grammars [9] can be used as declarative specifications of the dependence among computations at the nodes of a tree [13]. In their original form, however, they were difficult to write and to maintain. Moreover, early translators for attribute grammars generated evaluation modules with poor performance in both space and time. These problems have now been overcome. Modern attribute grammar tools, such as LIGA [7], generate modules whose efficiency is within a few percent of hand code from declarative specifications that are modular and reusable.

Here is a typical attribute grammar specification of computations at a node representing a C `while` statement:

```
RULE While:  it_statement ::= 'while' '(' exp ')' statement
COMPUTE
  it_statement.Ptg=PTGWhile(exp.Ptg, statement.Ptg, exp.Target, PTGLabel(NewPtgInt()));
  exp.Target=PTGLabel(NewPtgInt());
END;
```

This specification describes the computations involved in constructing a text fragment, using the module generated from the specification in the last section. Assume that `NewPtgInt` is defined operationally by the user, and yields a `PTGNode` representing the next integer each time it is invoked. `PTGLabel` might then be generated from a declarative specification like `Label:  "L"$` that constructs a valid assembly language label from an integer.

The dependence among the text fragments is defined by the `Ptg` attributes attached to the symbols in the rule. A label is generated by `PTGLabel(NewPtgInt())`, and sent down the expression tree by assigning it to `exp.Target`. This label will be incorporated into the text fragment representing the translation of the expression, which will be assigned to `exp.Ptg` by code generated from some rule with `exp` on the left-hand side. The value of the `Ptg` attribute of the `it_statement` on the left-hand side of the `While` rule then depends on that fragment, the fragment representing the translation of the statement controlled by the `while` (`statement.Ptg`), and two generated labels.

Clearly the `While` rule must have other computations associated with it. They could be added to the specification given above, but it would probably be better to place them in other files with related computations at other nodes. The specification given above would then be in a file with computations involving text fragment construction at nodes representing expressions and other statements. A tool like LIGA would then read all of the files, combining all of the computations for each individual rule, to obtain the complete attribute grammar.

The computations at a node appear in an attribute grammar as function applications; the tool that translates them does not recognize any operators like `+` or `==`. Modules must be provided to carry out all of the computations specified. When the computations are standard, like those needed to implement scope rules, they can be provided by library modules [8]. Computations that solve standard problems with special data structures, like those needed to maintain the definition table or construct text fragments, can be provided by modules generated by tools. Computations special to the application must be provided by modules written by the user. In any case, however, these modules need deal only with *local* computations. The tree traversal strategy and intermediate storage mechanisms are handled by the attribute grammar tool.

An attribute grammar tool can also generate a module to construct the tree described by the rules. For example, LIGA generates a module that exports a function for each rule and a function for each non-literal terminal symbol. Rule functions are invoked with one argument defining the source text coordinates and one argument for each non-literal symbol on the right-hand side of the rule. The latter arguments are pointers to tree nodes, and the function yields a pointer to the tree node representing the rule with the argument nodes as children. The name of the function is the name of the rule, prefixed by `Mk`. Thus the LIGA-generated tree construction module will export a rule `MkWhile` with three arguments for the rule described above.

Functions generated for non-literal terminal symbols are invoked with one argument defining the source text coordinates and one argument giving a single integer-valued *intrinsic attribute*.

3 The Eli System

The last section indicated that tools were available to generate all of the modules of Figure 1. It also pointed out that these modules depend upon one another in various ways. For example, the structural analysis module generated from a regular expression and a context-free grammar must invoke the functions exported by the tree construction module generated from an attribute grammar. In effect, there is a large amount of "glue" holding the modules of Figure 1 together.

If we treat the construction of the compiler as a single problem, rather than as a collection of module generation problems, then the nature of the interactions among the modules can often be deduced from their specifications. For example, the context-free grammar for C contains a rule `it_statement: 'while' '(' exp ')' statement`, and it is at precisely the reduction of this rule that the generated parser should invoke `MkWhile`. If we were using YACC to generate the parser module, we would write a a rule something like this in the context-free grammar:

```
it_statement:  WHILE '(' exp ')' statement {$$ = MkWhile($3,$5);};
```

A tool that had access to both the context-free grammar and the attribute grammar could easily deduce the action `{$$ = MkWhile($3,$5);}` by comparing the signatures of the two rules, and hence absolve the user of the need to add that action to the grammar.

Another example of the advantage of looking at the problem as a whole comes from the observation that no computations need be carried out at many of the nodes of the tree. Suppose that the user simply did not write rules for nodes at which no computation is required. The resulting attribute grammar would be a collection of unrelated fragments, and would be rejected by LIGA. A tool with access to both the context-free grammar and the attribute grammar could supply the missing rules by comparing the two. This represents a considerable saving for the user in a large grammar.

Domain-specific programming environments are systems that embody an understanding of certain classes of problems. They reduce the burden on the programmer by taking over standard tasks needed to solve problems in those classes. The programmer is then free to concentrate on the aspects of the problem that distinguish one instance from another, rather than having to deal with those that are common.

Eli is a domain-specific programming environment that embodies an understanding of the class of problems that involve analyzing text and then translating or interpreting it [5]. Those problems are solved by processors having the structure shown in Figure 1. Thus Eli contains tools that accept declarative specifications and generate modules to implement Figure 1. It also contains tools that analyze the relationships among those specifications and provide the "glue" to hold the modules together. Planner tools decide when certain of the blocks of Figure 1 are unnecessary, and change the manufacturing process accordingly.

An Eli user is unaware of the individual tools and their interactions. The user requests a *product*, such as an executable version of the compiler, and provides a set of specifications (which may be a mixture of declarative and operational). Eli examines the kinds of specifications, determines how to manufacture the desired product, and then does so. It maintains a cache of intermediate and final products so that the cost of manufacture after a change in specifications or requested product is minimized.

4 Example

The example is taken from a paper about the utility of application generators [3]. It is very simple, but it involves the specification of a complete translator. Another example, illustrating a larger problem, appeared in SIGPLAN Notices [14].

Jackson Structured Programming (JSP) is the design methodology used for this example [1,6]. JSP uses diagrams to relate the structure of the input file to the structure of the output file, and places operations according to that relationship. It is well suited to developing attribute grammar specifications, because it directly addresses the problem of deciding which computations should be associated with which nodes in a tree structure.

This section contains a *complete* specification of the program. The file used to produce this paper was also used to produce the generator. It is available (with appropriate instructions) at ftp.cs.colorado.edu, in file /pub/cs/distribs/eli/Examples/usenix.tar.Z. Be sure to use Eli3.5 or a later version to run it.

4.1 Statement of the problem to be solved

You need to classify a collection of several thousand words. There are no duplicate words in any class, but a given word may belong to more than one class. The classes have arbitrary names, and no two classes may have the same name.

A C program will manipulate the words and classes. Because of the application, classes will be relatively fluid. The customer expects that new words and classes will be added, and the classification of existing words changed, on the basis of experience and changing requirements. Nevertheless, the system design requires that the data be built into the C program at compile time.

The system designers have settled on an internal representation of the data involving an array for each class containing the words in that class as strings, an array containing the class names as strings, and an array containing the sizes of the classes as integers. The number of classes is also given. All of these data items are to be specified in a single header file. Here is an example of such a header file for a simple classification:

output[1] ≡

```
{int NumberOfSets = 3;

 char *NameOfSet[] = {
   "colors",
   "bugs",
   "verbs"};

 int SizeOfSet[] = {
   3,
   5,
   4};

 char *SetOf_colors[] = {
   "red",
   "blue",
   "green"};

 char *SetOf_bugs[] = {
   "ant",
   "spider",
   "fly",
   "moth",
   "bee"};

 char *SetOf_verbs[] = {
   "crawl",
   "walk",
   "run",
   "fly"};

 char **ValuesOfSet[] = {
   SetOf_colors,
   SetOf_bugs,
   SetOf_verbs};
 }
```

This macro is attached to an output file.

Although the meaning of the internal representation is straightforward, it is quite clear that making the necessary alterations will be a tedious and error-prone process. Any change requires compatible modifications

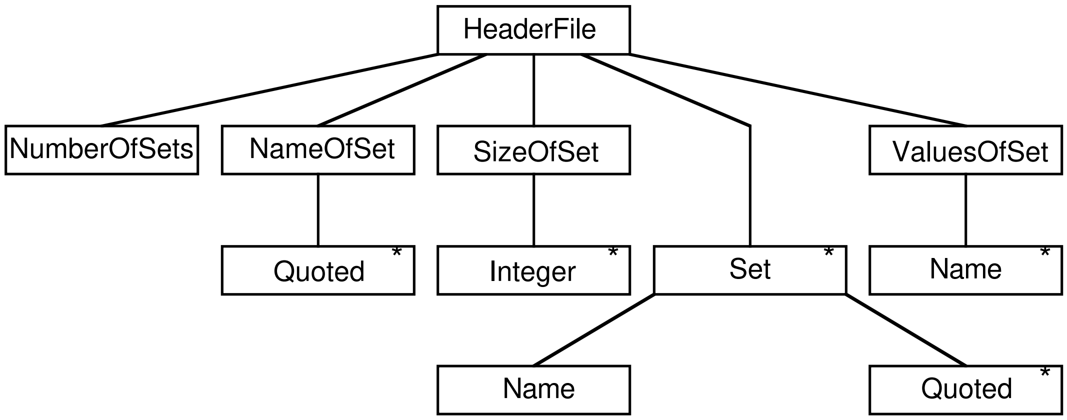

Figure 2: Structure of the Output File

of several arrays. For example, moving a word from one class to another means not only cutting and pasting the word itself, but also changing the sizes of both classes.

It would be simpler and safer to define the classification with a notation ideally suited to that task, and generate the header file from that definition. Here is an example of an obvious notation, defining the classification represented by the header file given above:

input[2] ≡
```
{colors{red blue green}
 bugs{ant spider fly moth bee}
 verbs{crawl walk run fly}
 }
```
This macro is attached to an output file.

4.2 Structure of the Output File

The header file to be output has the structure shown in Figure 2. Each box represents a component of the kind named in the box. The children of a box, taken in order, constitute the sub-components of that box. If a box contains an asterisk, it represents an arbitrary number of components of the given kind. You should convince yourself that Figure 2 does, indeed, describe the structure of the header file presented above.

We can describe the text associated with each node of Figure 2 by the following declarative specification:

OutputText.ptg[3] ≡
```
{HeaderFile:
   "int NumberOfSets = " $/*integer*/ ";\n\n"
   "char *NameOfSet[] = {\n"
   "  " $/*list of set names*/ "};\n\n"
   "int SizeOfSet[] = {\n"
   "  " $/*list of set sizes*/ "};\n\n"
   $/*sequence of sets*/
   "char **ValuesOfSet[] = {\n"
   "  " $/*list of set names*/ "};"

 Set:
   "char *" $/*set name*/ "[] = {\n"
   "  " $/*list of set elements*/ "};\n\n"

 Name:
   "SetOf_" $
```

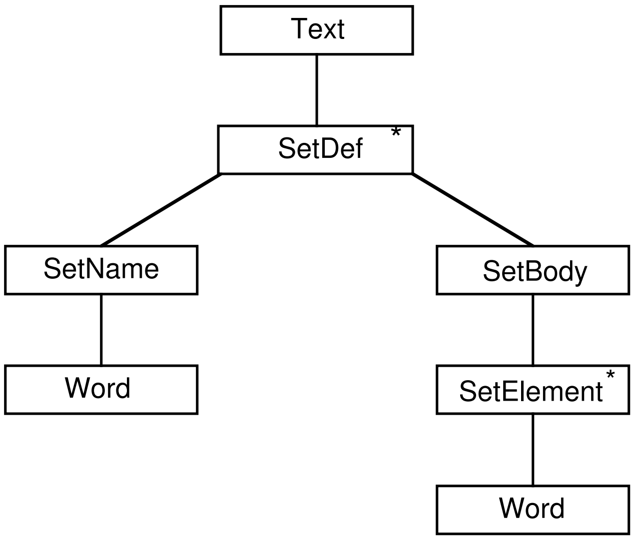

Figure 3: Structure of the Input File

```
Quoted:
  "\"" $ "\""

List:
  $ ",\n  " $

Seq:
  $ $
}
```

This macro is attached to an output file.

From this declarative specification, Eli will generate a module that exports functions named `PTGHeaderFile`, `PTGSet`, `PTGName`, `PTGQuoted`, `PTGList` and `PTGSeq`. `PTGSet` (for example) would be called with two arguments, one a `PTGNode` representing the name of the set and the other a `PTGNode` representing the list of quoted elements.

In addition to these functions, the module exports an output function `PTGOut` that takes a `PTGNode` as its argument and writes the text represented by that `PTGNode` on standard output. (Other output functions are also exported by the module.)

4.3 Structure of the Input File

Figure 3 gives the structure of the input file: an arbitrary number of set definitions, each consisting of a set name and a set body. The set body contains an arbitrary number of set elements.

Figure 3 reflects the problem requirement that there be only one occurrence of a given word in a given class. This sort of requirement is very common in translation problems. It involves recognition of regions and specific entities within those regions. For example, a `SetBody` is a region and a `SetElement` is a specific entity within that region. The requirement is that no `SetElement` appear more than once in any `SetBody`. Clearly a `SetBody` node must appear in Figure 3 in order to allow us to describe this requirement.

The structure of the input file can be described by the following context-free grammar:

InputText.con[4] ≡

```
{text: defs .
 defs: SetDef / defs SetDef .
 SetDef: SetName '{' SetBody '}' .
 SetName: Word .
```

```
SetBody: elements .
elements: SetElement / elements SetElement .
SetElement: Word .
}
```

This macro is attached to an output file.

The terminal symbols of the grammar are the literal braces and the nonliteral symbol `Word`. Eli can easily deduce that the braces are significant characters, but we must provide a definition of the significant character sequences that could make up a `Word`. We must also describe how to capture the significant information in a word for further processing.

Eli normally assumes that white space characters (space, tab and newline) are not significant. If we want to provide a facility for commenting a classification then we must additionally define the form of a comment and specify that character sequences having this form are not significant.

Here is one possible description of the non-literal character sequences:

InputText.gla[5] ≡

```
{Word:   $[a-zA-Z]+       [mkidn]
         C_COMMENT
}
```

This macro is attached to an output file.

The first line defines a `Word` by the regular expression `[a-zA-Z]+` (regular expressions are introduced by `$` and terminated by white space) [4]. Whenever such a sequence is recognized in the input text, `mkidn` is invoked to capture the significant information represented by the sequence. This processor (available in Eli's library) associates an integer with the recognized sequence, and arranges for that integer to become the value of the intrinsic attribute of the leaf representing the `Word`. If two character sequences recognized as words are identical, `mkidn` will associate the same integer with them; distinct sequences get different integers.

The second line of the specification does not begin with a symbol followed by `:`, which indicates that the character sequences it describes are not significant. It uses a *canned description* from Eli's library to describe character sequences taking the form of C comments. Thus any character sequence taking the form of a C comment will be ignored in the input text read by the generated program.

4.4 Placement of Tree Computations

Figure 4 shows the structure obtained by merging Figures 2 and 3. The tree computations that check the requirement of unique names and carry out the translation are attached to the nodes of this structure.

First consider the computation to check that set names are unique over the text and set elements are unique over their set. This is a standard compiler task involving nested scopes, and a solution is available in Eli's library. All that a user need do to make use of this solution is to instantiate the appropriate libraries and specify that their computations be inherited by the proper symbols in Figure 4.

The translation task is the essence of each compilation problem, and hence no complete standard solution can be provided. Standard solutions are, however, available for translating common data objects like strings and integers into `PTGNode`s that can be passed to the output tree construction functions. The remainder of this section gives the details of these computations.

4.4.1 Verifying unique set names and elements

To use the standard solution for this problem, the user must request that two generic libraries be instantiated:

Unique.specs[6] ≡

```
{$/Tool/lib/Name/Nest.gnrc :inst
 $/Tool/lib/Name/Unique.gnrc :inst
}
```

This macro is attached to an output file.

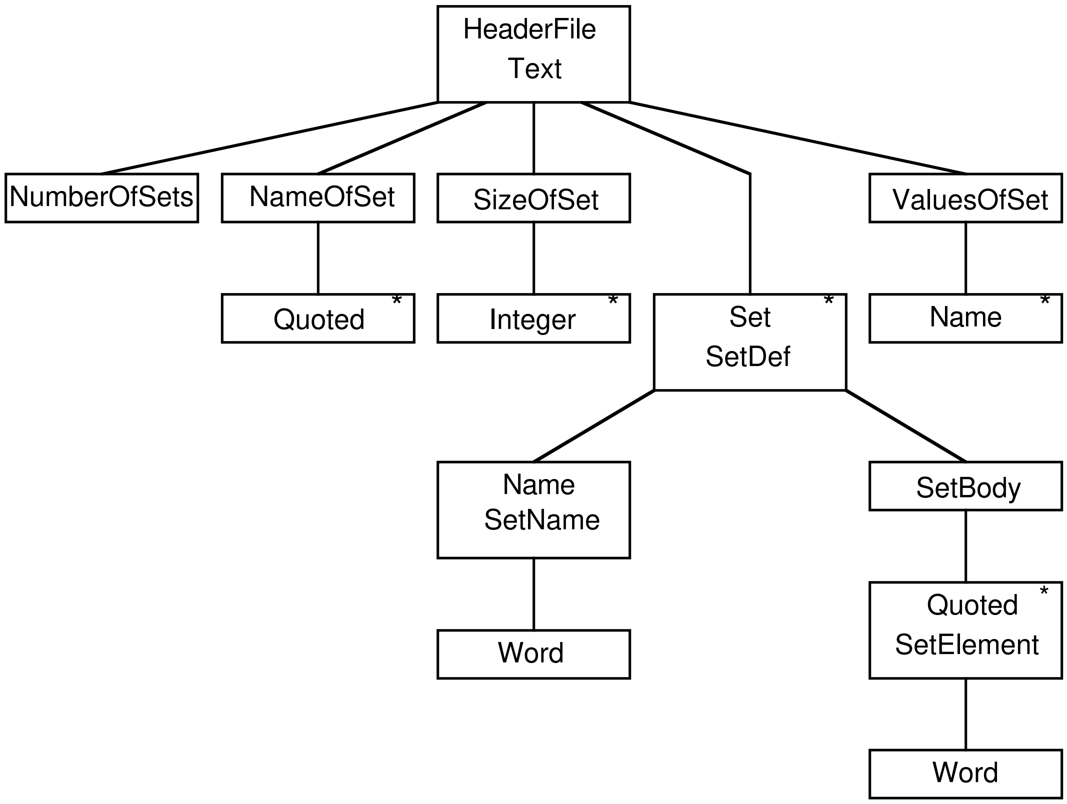

Figure 4: Merged Input and Output Structures

The `Nest` module provides the concept of nested regions containing entities distinguished by integer intrinsic attributes, while the `Unique` module provides the concept of an error for entities that appear more than once in a region.

An attribute grammar fragment is used to connect the concepts provided by these two modules to the nodes of Figure 4:

Unique.lido[7] ≡

```
{ATTR Sym: int;

SYMBOL Entity INHERITS IdDefNest, IdDefUnique COMPUTE
  SYNT.Sym=CONSTITUENT Word.Sym;
END;
SYMBOL SetName INHERITS Entity END;
SYMBOL SetElement INHERITS Entity END;

SYMBOL text INHERITS RootNest, RangeUnique END;
SYMBOL SetBody INHERITS RangeNest, RangeUnique END;
}
```

This macro is attached to an output file.

A `SetName` must be unique over the region lying outside of all sets, while a `SetElement` must be unique over the set in which it appears. Instead of duplicating the computations necessary for checking uniqueness at each of these symbols, we can define a new symbol (`Entity`) and associate the computations with it. Then we specify that the symbols `SetName` and `SetElement` inherit `Entity`'s computations.

The computations needed for a definition within a region are associated with the symbol `IdDefNest` exported by the `Nest` module, while the computations verifying that such a definition is unique are associated with the symbol `IdDefUnique`, exported by the `Unique` module. `Entity` inherits those computations from these two symbols.

Each `Entity` (either a `SetName` or a `SetElement`) consists of a single `Word`, which has the intrinsic attribute computed by `mkidn` when that `Word` was recognized. Eli allows any terminal to have a single integer-valued attribute whose value is never set, and equates that attribute with the single intrinsic attribute computed by the scanner. Conventionally, the name `Sym` is given to this attribute. `SYNT.Sym=CONSTITUENT Word.Sym` states that the `Sym` attribute of an `Entity` should be the same as the `Sym` attribute of the `Word` that is its descendent in the tree. (`SYNT` defines the computed attribute as a synthesized attribute [16] of the symbol with which the computation is associated – `Entity` in this case.)

Computations needed at the node representing the outermost region are associated with the symbol `RootNest`, and those needed at nodes representing inner regions are associated with the symbol `RangeNest`. Both of these symbols are exported by the `Nest` module. Computations needed at a node representing a region within which entities must be unique are associated with the symbol `RangeUnique`, exported by the `Unique` module.

4.4.2 Translating

The `LeafPtg` library provides computations that translate simple data objects into `PTGNode`s for use in building structured text:

LeafPtg.specs[8] ≡
```
{$/Tool/lib/Tech/LeafPtg.gnrc :inst
}
```
This macro is attached to an output file.

Computations needed to create a `PTGNode` representing a string from an attribute named `Sym` created by `mkidn` are associated with the symbol `IdPtg` exported by the `LeafPtg` library. The `PTGNode` created is the value of the `Ptg` attribute of that symbol. Thus every `SetName` and `SetElement` node in Figure 4 can be provided with a `Ptg` attribute whose value is the string representing that node in the input text by the attribute grammar fragment:

Word.lido[9] ≡
```
{ATTR Ptg: PTGNode;
 SYMBOL Entity INHERITS IdPtg END;
}
```
This macro is attached to an output file.

(Remember that both `SetName` and `SetElement` inherit the computations of `Entity`.)

Note that the boxes in the third row of Figure 4, all of which contain asterisks, all represent the same number of occurrences. In other words, there is one `Quoted`, one `Integer` and one `Name` for each `Set`. Also, the diagram states directly that there is one `SetName` for each `Set`. These are important insights, because they show us where the information for the translation comes from.

Figure 4 associates translation computations with only two nodes other than `SetName` and `SetElement`: The set size and set definition must be computed at `SetDef`, and the header file must be constructed and output at `text`. Each of these computations involves gathering information from descendants, in the form of either lists or totals.

Because gathering information from descendants is a very common operation, a special notation for it is provided: `CONSTITUENTS`. A `CONSTITUENTS` expression specifies an attribute of a descendant that acts as the ultimate source of the information, and a process for combining that information from all of the descendants possessing it. This process is described by a clause of the form `WITH (Type, Combine, Convert, Create)`. Here `Type` is the data type of the gathered information, while `Combine`, `Convert` and `Create` are functions yielding objects of that data type. `Combine` takes two `Type` objects and yields the result of combining them. `Convert` takes one object of the type of the attribute specified by the `CONSTITUENTS` expression and converts it to an appropriate `Type` object. `Create` creates an appropriate `Type` object from nothing.

The computations associated with `text` and `SetDef` both use `CONSTITUENTS` expressions heavily:

Build.lido[10] ≡

```
{ATTR Size: int;

 SYMBOL text COMPUTE
   PTGOut(
     PTGHeaderFile(
       PTGNumb(CONSTITUENTS SetDef.Ptg WITH (int, ADD, ONE, ZERO)),
       CONSTITUENTS SetName.Ptg WITH (PTGNode, PTGList, PTGQuoted, PTGNull),
       CONSTITUENTS SetDef.Size WITH (PTGNode, PTGList, PTGNumb, PTGNull),
       CONSTITUENTS SetDef.Ptg WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull),
       CONSTITUENTS SetName.Ptg WITH (PTGNode, PTGList, PTGName, PTGNull)));
 END;

 SYMBOL SetDef COMPUTE
   SYNT.Ptg=
     PTGSet(
       PTGName(CONSTITUENT SetName.Ptg),
       CONSTITUENTS SetElement.Ptg WITH (PTGNode, PTGList, PTGQuoted, PTGNull));
   SYNT.Size=CONSTITUENTS SetElement.Ptg WITH (int, ADD, ONE, ZERO);
 END;
 }
```

This macro is attached to an output file.

Consider the computation of `SetDef.Size`. This computation gathers information from the `Ptg` attributes of all of the `SetElement` descendants of the `SetDef` node. It gathers this information by applying `ONE` to every such attribute and passing the type-`int` result to that node's parent. At every node with more than one descendant, the computation applies `ADD` pairwise to the results from those descendants. If one of those descendants has no `SetElement` nodes as descendants, its result is obtained by evaluating `ZERO`.

`ADD` is an integer addition from Eli's library, while `ZERO` and `ONE` are defined operationally by the user as follows:

Operations.head[11] ≡

```
{#define ZERO(x) 0
 #define ONE(x) 1
 }
```

This macro is attached to an output file.

Thus the effect of the expression `CONSTITUENTS SetElement.Ptg WITH (int, ADD, ONE, ZERO)` is to count the number of `SetElement` descendants of the `SetDef` node.

`CONSTITUENTS SetElement.Ptg WITH (PTGNode, PTGList, PTGQuoted, PTGNull)` can be analyzed similarly. `PTGNull` is a function exported by the structured text creation module. It yields `PTGNULL`, a distinguished `PTGNode` that represents the empty string.

The expression `CONSTITUENT SetName.Ptg` is valid if and only if the current node has exactly one descendant that is a `SetName` node. It's value is the `Ptg` attribute of that node. Note that because there is exactly one such descendant, no combining of information is required.

Finally, `PTGNumb` is a function exported by the `LeafPtg` library module. It accepts an argument of type `int` and delivers a `PTGNode` representing a sequence of digits giving the value of that argument.

Acknowledgements The Eli system is a joint project of the University of Colorado and the University of Paderborn. Its development has been supported by the US Army Research Office under grant DAAL03-92-G-0158, and by the government of Nordrhein-Westfalen through the SofTec-Cooperation.

References

1. Cameron, J., *JSP & JSD: The Jackson Approach to Structured Programming*, IEEE Press, Montvale, NJ, 1989.

2. Chomsky, N., "Three Models for the Description of Language," *IRE Transactions on Information Theory* IT-2 (1956), 113–124.

3. Cleaveland, J. C., "Building Application Generators," *IEEE Software* 5 (July 1988), 25–33.

4. Gray, R. W., "A Generator for Lexical Analyzers That Programmers Can Use," *Proceedings USENIX Conference* (June 1988).

5. Gray, R. W., Heuring, V. P., Levi, S. P., Sloane, A. M. & Waite, W. M., "Eli: A Complete, Flexible Compiler Construction System," *Communications of the ACM* 35 (February 1992), 121–131.

6. Hughes, J. W., "A Formalization and Explication of the Michael Jackson Method of Program Design," *Software - Practice & Experience* 9 (1979), 191–202.

7. Kastens, U., "Attribute Grammars as a Specification Method," in *Proceedings of the International Summer School on Attribute Grammars, Application and Systems*, Lecture Notes in Computer Science #545, Springer Verlag, New York–Heidelberg–Berlin, 1991, 16–47.

8. Kastens, U. & Waite, W. M., "An Abstract Data Type for Name Analysis," *Acta Informatica* 28 (1991), 539–558.

9. Knuth, D. E., "Semantics of Context-Free Languages," *Mathematical Systems Theory* 2 (June 1968), 127–146.

10. McClure, R. M., "An Appraisal of Compiler Technology," in *Spring Joint Computer Conference*, AFIPS Conference Proceedings #40, AFIPS Press, Montvale, NJ, 1972.

11. "Revised Report on the Algorithmic Language ALGOL 60," *Communications of the ACM* 6 (January 1963), 1–17.

12. Rutishauser, H., *Automatische Rechenplanfertigung bei Programm-gesteuerten Rechenmaschinen*, Mitteilungen aus dem Institut für Angewandte Mathematik der ETH-Zürich #3, Birkhaüser, Basel, 1952.

13. Waite, W. M., "Use of Attribute Grammars in Compiler Construction," in *Attribute Grammars and their Applications*, Pierre Deransart & Martin Jourdan, eds., Lecture Notes in Computer Science #41, Springer Verlag, Berlin, 1990, 255–265.

14. Waite, W. M., "An Executable Language Definition," *SIGPLAN Notices* 28 (February 1993), 28–40.

15. Waite, W. M. & Carter, L. R., *An Introduction to Compiler Construction*, HarperCollins, New York, 1993.

16. Waite, W. M. & Goos, G., *Compiler Construction*, Springer Verlag, New York, NY, 1984.