Design, Implementation, Use, and Evaluation of Ox: An Attribute-Grammar Compiling System based on Yacc, Lex, and C

Kurt M. Bischoff
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Programming Languages and Compilers Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/23

This Article is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Design, Implementation, Use, and Evaluation of Ox: An Attribute-Grammar Compiling System based on Yacc, Lex, and C

Abstract
Ox generalizes the function of Yacc in the way that attribute grammars generalize context-free grammars. Ordinary Yacc and Lex specifications may be augmented with definitions of synthesized and inherited attributes written in C syntax. From these specifications, Ox generates a program that builds and decorates attributed parse trees. Ox accepts a most general class of attribute grammars. The user may specify postdecoration traversals for easy ordering of side effects such as code generation. Ox handles the tedious and error-prone details of writing code for parse-tree management, so its use eases problems of security and maintainability associated with that aspect of translator development. The translators generated by Ox use internal memory management that is often much faster than the common technique of calling malloc once for each parse-tree node. Ox is a Yacc/Lex/C preprocessor, and is designed to bring attribute grammars closer to the mainstream of Unix-based language development. Ox inherits all of the familiar syntax and semantics of Yacc, Lex, and C. It is relatively easy to convert programs between Ox code and “pure Yacc/Lex/C” code. Ox has been used to build a compiler for a small (eighty grammar rules) block-structured imperative programming language. This paper considers Ox’s design, implementation, and use, evaluates the performance of Ox-generated evaluators, and makes recommendations for improvements in Ox.

Disciplines
Programming Languages and Compilers

This article is available at Iowa State University Digital Repository: http://lib.dr.iastate.edu/cs_techreports/23
Design, Implementation, Use, and Evaluation of Ox: An Attribute-Grammar Compiling System based on Yacc, Lex, and C

by Kurt M. Bischoff
TR#92-31
December, 1992

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011

©1992 Kurt M. Bischoff
Contents

1 General Introduction and User Manual  5
  1.1 Overview of Use .......................... 5
  1.2 Preliminary ................................ 6
  1.3 Attribute Declarations ..................... 7
    1.3.1 Semantics of Attribute Declarations  8
  1.4 Rules and Attribute Occurrences ............. 8
  1.5 Ox Attribute Definitions ..................... 9
    1.5.1 Inherited vs. Synthesized Attributes  9
    1.5.2 Attribute Reference Sections in the Y-file 10
    1.5.3 Attribute Reference Sections in the L-file(s) 13
    1.5.4 Cycles ................................ 14
  1.6 Translation into C Code ....................... 14
  1.7 Temporal Behavior of the Ox-generated Evaluator 15
    1.7.1 Stack Operations ....................... 15
    1.7.2 Placement of Generated Code ............. 15
    1.7.3 Decoration and the Ready Set ............. 16
  1.8 Programming Style .......................... 17
  1.9 Postdecoration Traversals .................... 17
    1.9.1 Example: Infix to Prefix Translation 18
    1.9.2 General Description ................... 20
  1.10 Macros .................................. 24
    1.10.1 Macro Definitions ................... 24
    1.10.2 Macro Uses .......................... 25
    1.10.3 Example ............................. 25
  1.11 Shell Command Sequences for Evaluator Development 26
    1.11.1 Conventions of Naming Ox Output Files 26
1.11.2 Review: Combining the Outputs of Yacc and Lex . . . . 26
1.11.3 Combined Use of Ox, Yacc, and Lex . . . . . . . . . 27
1.11.4 Typical Command Sequences . . . . . . . . . . . . . . 27

1.12 Other Points and Features . . . . . . . . . . . . . . . . 28
1.12.1 Error Recovery . . . . . . . . . . . . . . . . . . . . . 28
1.12.2 Memory Alignment . . . . . . . . . . . . . . . . . . . 28
1.12.3 Stripping Ox Constructs . . . . . . . . . . . . . . . . 28
1.12.4 Preventing Execution of Attribute Definition Code . . . 29
1.12.5 Control of Storage Allocation in the Generated Evaluator 29
1.12.6 Parse Tree Statistics . . . . . . . . . . . . . . . . . . . 30
1.12.7 Adjusting the Sizes of Ox’s Data Structures . . . . . . 31

1.13 Example: An Integer Calculator . . . . . . . . . . . . . . 31
1.14 Example: A Binary Number Translator . . . . . . . . . . 33
1.15 Example: Translation from Infix to Postfix and Prefix . . . 37

2 General Design Criteria and Comparison with Other Systems 40
2.1 Generalization of the Function of Yacc . . . . . . . . . . . 40
2.2 Generality of the Class of Attribute Grammars Accepted . . 41
2.3 Ox as a Preprocessor . . . . . . . . . . . . . . . . . . . . . 41
2.4 Portability . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
2.5 Competing Systems . . . . . . . . . . . . . . . . . . . . . . 42
2.5.1 The Delft System . . . . . . . . . . . . . . . . . . . . . 42
2.6 Performance . . . . . . . . . . . . . . . . . . . . . . . . . . 42

3 Design of the Ox-generated Evaluator . . . . . . . . . . . . . 44
3.1 Topological Sort Using Dependee Counts . . . . . . . . . . . 44
3.2 Static Phase of the Topological Sort . . . . . . . . . . . . . 45
3.3 Parse-Tree Nodes . . . . . . . . . . . . . . . . . . . . . . . . 47
3.4 Items on the Parse-Tree Stack . . . . . . . . . . . . . . . . 48
3.5 Addressing Attribute Instances and Dependee Counts . . . 48
3.6 Static Data Structures of the Generated Evaluator . . . . . . 49
3.6.1 Table of Dependee-Count-Initialization Lists . . . . . . . 49
3.6.2 Table of Decoration-time-executed Code Fragments . . . 49
3.6.3 Table of Lists of Dependents . . . . . . . . . . . . . . . 49
3.7 Solved Instances Belonging to Parentless Nonterminal Nodes . 50
3.8 Operations on the Parse-Tree Stack . . . . . . . . . . . . . . 50
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1.2</td>
<td>Eliminating Attribute-less Leaf Nodes</td>
<td>72</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Removing Indirection from Parse-Tree Nodes</td>
<td>72</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Calculation of Space Savings</td>
<td>73</td>
</tr>
<tr>
<td>6.2</td>
<td>Grammar-wide Semantic Analysis</td>
<td>74</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Static Cirularity Test</td>
<td>74</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Solving More Instances at Reduction Time</td>
<td>74</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Global Static Topological Sorting</td>
<td>74</td>
</tr>
<tr>
<td>6.3</td>
<td>Extensions to Ox Syntax and Semantics</td>
<td>75</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Dependees for Synthesized Occurrences of Tokens</td>
<td>75</td>
</tr>
<tr>
<td>6.3.2</td>
<td>L-file(s) not Based on Lex</td>
<td>76</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Defaults for Attribute Definitions</td>
<td>76</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Automatic Construction of Syntax Trees</td>
<td>76</td>
</tr>
<tr>
<td>6.3.5</td>
<td>Traversals at Reduction Time</td>
<td>77</td>
</tr>
<tr>
<td>6.3.6</td>
<td>Repetition of Traversals</td>
<td>77</td>
</tr>
<tr>
<td>6.4</td>
<td>Facility for Interactive Debugging</td>
<td>77</td>
</tr>
<tr>
<td>6.5</td>
<td>Support for Other Languages</td>
<td>78</td>
</tr>
<tr>
<td>6.6</td>
<td>Multiprocessing</td>
<td>78</td>
</tr>
</tbody>
</table>
Chapter 1

General Introduction and User Manual

1.1 Overview of Use

Lex and Yacc are powerful and widely-used tools for the automatic generation of language recognizers. Lex accepts a set of user-written regular expressions and writes a C program that performs lexical analysis according to those expressions. Yacc translates user-written grammar rules into C source code for a syntax analyzer. While they afford “hooks” for execution of hand-coded C-language semantic actions, Lex and Yacc provide little other facility for automatic implementation of language semantics.

Attributed parse trees are often used as data structures in evaluators for languages. Often the language implementer hand-crafts code for building, traversing, and evaluating attributes of parse trees, and for parse-tree-related memory management. A Yacc specification defines a context-free language and a mapping from the set of legal sentences to the set of parse trees, but code for parse-tree management is not generated automatically by Yacc.

The Ox 1 user specifies a language using the familiar languages of Lex and Yacc, or takes an existing Lex/Yacc specification, and adds semantics to the language by augmenting the specification files with declarations and

---

1 The name “Ox” originated as a homophone for an acronym for “An Attribute Grammar Compiling System”. It was noticed later that every yak is an ox and that Ox generalizes the function of Yacc.
definitions of typed attributes of parse-tree nodes.

That specification constitutes an attribute grammar, and from it Ox automatically generates an evaluator written in Yacc, Lex, and C. For a given input, the evaluator builds a parse tree, determines an order of evaluation for attributes of the tree, and performs, for each attribute, the semantic action required to evaluate it. This parse tree is managed independently of any trees managed by hand-written C code, but information may be moved between the evaluator-managed tree and any global data structure.

Additionally, the Ox user may easily specify parse-tree traversals that are performed after evaluation of the tree’s attributes and that refer to those attributes. Such traversals greatly simplify tasks such as code generation and the gathering of compilation statistics.

The language designer is freed from the tedious and error-prone details of writing code for parse-tree management. Ox-generated evaluators use memory-management techniques that bring large time-efficiency gains over hand-built evaluators that use the common technique of calling malloc once for each parse-tree node. Also, Ox provides security by testing the definition for consistency and completeness, and the Ox-generated evaluator performs tests to ensure that a circular definition has not prevented evaluation of attributes.

Ox is a preprocessor that accepts two or more files, and translates these into files suitable for input to Lex and Yacc. With few exceptions, all Lex-input/Yacc-input pairs of files that constitute recognizers or translators are legal inputs to Ox. Thus much existing software is amenable to modification using Ox, and implementations that use Ox may be converted stepwise by hand to “pure” Lex/Yacc implementations. This makes Ox well-suited to language designers, experimenters, and implementers already familiar with Lex, Yacc, and C.

1.2 Preliminary

It is assumed that the reader is familiar with the use of Yacc [Johnson 75], Lex [Lesk 75], and C [KR 88]; Ox syntactic constructs are described mainly as augmentations of the languages accepted by those tools. Prior acquaintance with the basic ideas of attribute grammars (for instance, as found in [Waite 84] or [Aho 86]) is helpful.
An Ox input specification consists of at least two files: a syntactic specification (which resembles a Yacc input specification and is called a Y-file) that Ox translates into a Yacc input specification, and one or more lexical specifications (which resemble Lex input specifications and are called L-files) that Ox translates into Lex input specifications. Usually there is exactly one L-file, but an evaluator that uses more than one scanner [Lesk 75] may be constructed by submitting to Ox more than one L-file. This manual presents descriptions of the Ox-specific constructs that may appear in these files, as well as pertinent underlying concepts. These constructs are illustrated mainly by using fragments of three examples of Ox input specifications, the complete texts of which appear in sections 1.13, 1.14 and 1.15.

Within Ox-specific constructs, C-style comments may appear anywhere whitespace may appear. The global identifiers of Ox-generated C code, like those generated by Yacc and Lex, are prefixed by yy, so the Ox user can avoid name conflicts in the generated evaluator by abstaining from the use of global identifiers that begin with yy.

1.3 Attribute Declarations

As described in [Johnson 75], the declarations section of a Yacc input specification is the part that precedes the first %% mark, and in it the user may declare the start symbol, tokens, associativities, unions, C code sections, etc. The Y-file contains such a declarations section, and in it are permitted all of the constructs of Yacc declarations sections, as well as Ox attribute declarations. An attribute declaration consists of the reserved word @attributes followed by {, an attribute declaration list, }, and a list of grammar symbols. Suppose that a grammar has a symbol bitlist and the following attribute declaration:

```c
@attributes {float value; int scale,length;} bitlist
```

Then the Ox-generated evaluator, when building a parse-tree node labeled bitlist, allocates storage for a float named value and integers named scale and length.

An attribute declaration list (in the previous example, the part between curly braces) resembles a C structure declaration list. Digit strings and C-style identifiers, as well as the following characters and reserved words, arranged according to C syntax, are legal in attribute declaration lists:
Note that curly braces may not appear inside (and so structures and unions may not be declared inside) attribute declaration lists. Despite this, any fundamental or derived type permitted in a C program may be used as an attribute type specifier: Yacc input specifications often contain C code sections between `{` and `}`, and these are also permitted in Ox input specifications. Any type name given meaning by using `struct`, `union`, `typedef`, or `#define` in a previous C code section may be used as an attribute type specifier.

The list of grammar symbols following `{` is a possibly empty list of Yacc tokens (including character constants) and nonterminals, members of the list being separated by whitespace.

### 1.3.1 Semantics of Attribute Declarations

An attribute declaration informs Ox that each symbol in the grammar symbol list has attributes of the names and types appearing in the attribute declaration list. If `a` appears in the attribute declaration list and `s` appears in the grammar symbol list, then `a` is said to belong to `s` or to be an attribute of `s`. Each grammar symbol has its own attribute name space. When the evaluator creates a node labeled by one of the listed symbols, it allocates storage of the specified type for each of the named attributes. A storage location so allocated is called an attribute instance (concisely, an instance) in the parse tree. Instances may be said to belong to nodes.

### 1.4 Rules and Attribute Occurrences

Yacc grammar rules (productions), and the objects of return statements in Lex actions (each such object being a token), are here referred to generically as rules. Since Ox accepts the constructs of Yacc and Lex, and passes these unchanged, the corresponding constructs of Ox input specifications are also called rules. Each rule is viewed as a sequence of grammar symbols, the object of each return statement in a Lex action being a sequence consisting of a single grammar symbol. The leftmost symbol of a rule is called the left hand side (LHS), and the right hand side (RHS) is comprised of the rule’s other symbols. A symbol’s position in a rule together with an attribute of
that symbol constitute an *attribute occurrence* (concisely, an *occurrence*) in that rule. If the attribute in question is \( a \), the occurrence is said to be an occurrence of \( a \). Supposing the \@attributes declaration of section 1.3 and the rule:

\[
\text{num} : \text{bitlist DOT bitlist}
\]

the attribute occurrence *scale* of the leftmost appearance of \text{bitlist} is denoted in Ox code as \text{bitlist.0.scale}, while the attribute occurrence *scale* of the rightmost appearance of \text{bitlist} is denoted \text{bitlist.1.scale}.

In general, attribute occurrences are named by a grammar symbol, followed by a period, followed optionally by a non-negative decimal integer and another period, followed by the name of an attribute of that symbol. The integer and the second period are needed only when a given grammar symbol appears more than once in the rule, in which case those distinct appearances are numbered from left to right with consecutive increasing integers starting with 0. For a symbol \( X \) with an attribute \( a \), \( X.a \) is a synonym for \( X.0.a \).

A given rule and an attribute occurrence in that rule constitute an *attribute occurrence* in the grammar.

### 1.5 Ox Attribute Definitions

For each rule, the Ox user may provide an *attribute reference section*, delimited by \@{} and \@{}, and optionally containing definitions of attribute occurrences of the given rule. Attribute occurrences may be defined therein in terms of the rule’s other attribute occurrences and C code such as global variables, constants, macros, and function calls.

#### 1.5.1 Inherited vs. Synthesized Attributes

An attribute occurrence \( o \) of a rule \( R \) is *synthesized* if and only if

1. \( o \) is on the LHS of \( R \) and the attribute reference section of \( R \) contains a definition of \( o \), or
2. \( o \) is on the RHS of \( R \) and the attribute reference section of \( R \) contains no definition of \( o \).

An attribute occurrence \( o \) of a rule \( R \) is *inherited* if and only if
1. $o$ is on the LHS of $R$ and the attribute reference section of $R$ contains no definition of $o$, or

2. $o$ is on the RHS of $R$ and the attribute reference section of $R$ contains a definition of $o$.

An error message is issued if an attribute is found to have both synthesized and inherited occurrences in the grammar. An attribute is synthesized if and only if it has at least one occurrence, and its every occurrence is synthesized. An attribute is inherited if and only if it has at least one occurrence, and its every occurrence is inherited. It follows from the above that the grammar’s start symbol may have only synthesized attributes. Referring to returned tokens as rules emphasizes the equal status of tokens and non-terminals, inasmuch as each kind of symbol (except the start symbol) may have both synthesized and inherited attributes. Since each symbol has a distinct name space, same-named attributes of different symbols are different attributes, and may differ as to whether they are inherited or synthesized.

For each parse-tree node except the root node, two rules of the Ox input specification are of particular interest. The home rule is the rule applied at the node, i.e., the rule whose LHS is the label of the given node, and whose RHS symbols are the labels of the children of the node. The parent rule is the rule applied at the node’s parent. The attribute definition of a synthesized attribute instance of a given node is associated with the node’s home rule (i.e., it appears in the attribute reference section for that rule), and definitions of inherited attribute instances are similarly associated with the parent rule.

In a legal input specification, each attribute of a symbol appearing in a rule is either synthesized or inherited, but not both, so the definitions of all attributes “fit together” completely and without contradiction.

1.5.2 Attribute Reference Sections in the Y-file

The rules section of a Yacc file follows the first %% mark [Johnson 75], and contains the productions (rules) of the grammar. As mentioned above, the Ox user may augment each rule by an attribute reference section, each of which is delimited by $\{\text{ and }\}$, and which contains zero or more attribute definitions. When present, the attribute reference section is the last item
(other than a terminating semicolon) in a rule. Thus it does not precede any `Yacc` action or the `Yacc` reserved word `%prec` in the rule, and any following identifier must be the LHS of the next rule. Conceptually, an attribute definition has a dependence part and an evaluation part, but syntactically, the parts may be combined or separate. There are three modes of expression of attribute definitions, and different modes may be used within a single attribute reference section. Each attribute definition begins with a definition mode annunciator (@e, @i, or @m), and is terminated by another mode annunciator or by @).

**Explicit Mode**

In this, the most powerful and most verbose attribute definition mode, an attribute definition takes the form of @e (mnemonic for *explicit*) followed by a *dependency expression* (which expresses the dependency part of the definition) followed by an *evaluation expression* (which expresses the evaluation part). In the following example, the attribute reference section contains three attribute definitions, each expressed in the explicit mode:

```c
num : bitlist DOT bitlist
@{ @e num.value : bitlist.0.value bitlist.1.value;
   @num.value@ = @bitlist.0.value@ + @bitlist.1.value@;
   @e bitlist.0.scale : ;
   @bitlist.0.scale@ = 0 ;
   @e bitlist.1.scale : bitlist.1.length ;
   @bitlist.1.scale@ = -@bitlist.1.length@;
}@;
```

A dependency expression explicates constraints on the order of execution of evaluation expressions and is a non-empty list of attribute occurrences of the rule, followed by a colon, followed by a possibly empty list of attribute occurrences and a terminating semicolon. The occurrences to the left of the colon are said to *depend upon* (hence are called *dependents of*) those to the right, and are the occurrences *defined* in the given attribute definition. The occurrences to the right are *depend upon* by those on the left, and are called *dependents of* those on the left. An evaluation expression is basically a C code fragment that may contain attribute references, each of which is an attribute occurrence enclosed within @ symbols. Attribute references behave as C variables, and all of the usual C operators, such as those for arithmetical, logical,
and pointer operations, may be applied to them, as in a C program. The evaluation expression immediately follows the semicolon of the dependency expression.

The Ox-generated evaluator chooses an evaluation order such that the evaluation expressions for all of the dependees in the definition are executed before those of the dependents. Usually there is a single dependent in a given attribute definition, but in some cases, code may be made more compact by placing more than one attribute occurrence in a dependent list, thereby combining the definitions of those in the list. The evaluation expression is executed on behalf of the dependents taken as a set, rather than once for each dependent. This is known as solving the attribute instances corresponding to the occurrences in that set.

Implicit Mode

The implicit mode, which is the usual mode of expressing attribute definitions, syntactically combines the dependency part with the evaluation part. The following Ox code is equivalent to that of the preceding example.

```plaintext
num : bitlist DOT bitlist
  @i @num.value@ = @bitlist.0.value@ + @bitlist.1.value@;
  @i @bitlist.0.scale@ = 0;
  @i @bitlist.1.scale@ = -@bitlist.1.length@;
@}
;
```

In this mode, an attribute definition takes the form of @i followed by an evaluation expression. The mode announcator @i informs Ox that the definition has a single dependent, namely the first attribute occurrence referenced in the evaluation expression. The dependees in the definition consist of all other attribute occurrences referenced in the evaluation expression.

Mixed Mode

Mixed mode attribute definitions are announced by the reserved word @m, and there follow one or more dependents, a semicolon, and an evaluation expression. The occurrences referenced in the evaluation expression, except those that also appear between @m and the semicolon, are taken to be the dependees in the definition. Thus the dependents are given explicitly and
the dependees implicitly. The code in the following example has the same meaning as that in the previous two.

```c
num : bitlist DOT bitlist
    @{
        @m num.value;
        @num.value@ = @bitlist.0.value@ + @bitlist.1.value@;
        @i @bitlist.1.scale@ = - @bitlist.1.length@;
        @m bitlist.0.scale ; @bitlist.0.scale@ = 0;
    @} ;
```

1.5.3 Attribute Reference Sections in the L-file(s)

Definitions of inherited attributes of tokens are associated with rules appearing in the Y-file, while their synthesized attributes are defined in the L-file(s). Ox processes the Y-file before processing the L-file(s). If a given attribute occurrence of a token is not defined in the Y-file, then the attribute is taken to be synthesized.

Lexical rules are associated with return statements in Lex actions. After the terminating semicolon of each such statement, there may appear a possibly empty attribute reference section, delimited by @{ and @} in which are defined all of the synthesized attributes of the returned token.

The class of attribute grammars accepted by Ox is restricted only as follows: synthesized attributes of tokens do not have dependees. Attribute definitions in the L-file(s) can thus be written more simply than in the Y-file; each attribute occurrence is defined by referring to it in C code, exactly once in the attribute reference section associated with the return statement, as in the following example.

```c
[0-9]+       return(CONST); @{ sscanf(yytext,"%d",&@CONST.val@); @} ;
```

Thus mode declarations and dependency expressions are unnecessary in the L-file(s).

A difficulty arises in rules like

```c
return(*yytext);
```
and

```c
return(cond?TOKEN1:TOKEN2);
```
for which Ox cannot determine at evaluator-generation time which token will be returned. Ox issues a warning in such cases. In the first case, wherein no declared token or character constant is recognized in the returned expression, Ox assumes that the token returned has no attributes. In the second case, wherein more than one declared token or character constant is recognized, the node appended to the tree during evaluation is of the type of the declared token or character constant appearing first in the expression.

1.5.4 Cycles

It is easy to write an attribute grammar such that some attribute instance of some parse tree has a chain of dependencies that leads back to itself. Such a grammar is called circular, and such a chain of dependencies is called a cycle. For such a tree, there is an attribute instance that the evaluator cannot begin to solve until that instance has already been solved. A cycle also makes it impossible to solve any attribute instance that has a chain of dependencies leading to an instance involved in the cycle. Circularity is usually not intended by the evaluator designer. A general circularity test performed at evaluator-generation time would require exponential running time for some inputs [Jazayeri 75]. Polynomial-time tests for special kinds of non-circularity are known, but the present version of Ox deals with the problem by checking for cycles at evaluation time.

1.6 Translation into C Code

Ox translates attribute declarations into C structure declarations, with the attribute names appearing as structure members.

The evaluation expression of each attribute definition is copied verbatim into Ox’s output, except that attribute references are translated into parenthesized references to C variables.
1.7 Temporal Behavior of the Ox-generated Evaluator

1.7.1 Stack Operations

Inasmuch as an ordinary Yacc/Lex recognizer employs an LR parsing algorithm [Aho 86], each input entails a sequence of lookaheads, shifts, and reductions, and a stack of parser states is maintained. From ordinary Yacc/Lex source, Ox generates an evaluator whose yyparse goes through the same sequence of lookaheads, shifts, and reductions as does the yyparse of the ordinary Yacc/Lex recognizer.

The Ox-generated evaluator, in building a parse tree, maintains a stack of subtrees. The operations on the stack of subtrees are synchronized with the operations yyparse performs on its stack of parser states, except that operations involving the “marker nonterminals” (see [Johnson 75]) inserted into the grammar by Yacc are ignored.

The evaluator maintains its stack of subtrees as follows. Lookaheads coincide with calls to yylex. Just before a return is executed in a Lex action, an image of a leaf node is created in the evaluator’s lookahead buffer, and its synthesized attribute instances are solved and placed in that buffer. At each shift, a leaf node is created from the image in the lookahead buffer, and the subtree consisting of that leaf node is pushed onto the stack. At each reduction, zero or more subtrees are popped from the stack, and their roots become the children of a newly-created node, yielding a new subtree. The root of the new subtree is given a label to indicate the production being applied at the node, and the new subtree is pushed onto the stack. The parse tree is completed upon end of input together with reduction to the start symbol.

1.7.2 Placement of Generated Code

Code for parse-tree management and attribute evaluation is placed in Yacc and Lex actions in Ox’s output. If a given rule in the Y-file has an ordinary Yacc action, the Ox-generated code is placed after any programmer-supplied C code contained in the action. If a given rule in the Y-file lacks a Yacc action, an action is created, and the Ox-generated code is placed there. The
actions so created are introduced only at the ends of rules, so Yacc does not create a marker nonterminal for the action, and the LALR(1) property of the grammar is unaffected.

When an attribute reference section in an L-file contains definitions for more than one attribute occurrence, code for implementing those definitions is executed in the same order in which the definitions appear in that section.

For the attribute occurrences defined in the Y-file, Ox and the Ox-generated evaluator perform analyses to determine when to execute the code segment that evaluates a given attribute. The order of execution of the code segments associated with the definitions in a given attribute reference section is determined by the dependencies of the definitions, and is not necessarily related to the order of appearance of the definitions.

Some attribute occurrences, for example those that have no dependees, are evaluated as part of the Yacc action executed upon reduction by the associated production. Definitions of such occurrences are allowed to contain references to the Yacc pseudovariables $$, $1, $2, etc. If Ox determines that a given attribute occurrence cannot be evaluated at reduction time, and the definition refers to such a pseudovariable, Ox issues an error message.

### 1.7.3 Decoration and the Ready Set

The Ox-generated evaluator maintains a set of attribute instances that are ready to be solved, i.e., those whose every dependee has been solved, but which have not themselves been solved. During parsing of the input, it is possible to remove an attribute instance from this ready set, solve it, and then check whether the solving of that instance has caused any of its dependents to be ready to be solved. Instances that are thus made ready are then placed in the ready set. Repeating this process until the ready set is empty is known as decoration. Following a decoration, further parsing of the input may result in creation of parse-tree nodes and insertion of attribute instances into the ready set. Scheduling of decorations is performed automatically by the evaluator. Evaluation of a given syntactically-correct input involves at least one decoration, that performed after the final reduction to the start symbol.
1.8 Programming Style

Definitions of *attribute grammar*, (for instance those in [Lorho 88] and [Waite 84]) employ no notion of execution sequence. The usual *Ox* programming style involves defining synthesized attribute occurrences of tokens in terms of `yytext` and `yyleng` and other such data structures of the scanner. Then the attribute definitions of each production are written only in terms of constants and other attribute occurrences of that production. For a given sentence, the synthesized attribute instances of the tokens then completely determine the values of all attribute instances of the parse tree. The attribute instances of the root node are often of particular interest, and their definitions often contain code that copies their values to global C variables, so that they may be used in code executed after the return from `yyparse`.

Since attribute definitions in *Ox* code may contain any C code, the *Ox* programmer may deviate from the safe approach described above by using non-root attribute definitions that read or write global variables. Before attempting the use of side effects, the programmer should be familiar with the material of section 1.7.

Since the order of evaluation of attributes by the *Ox*-generated evaluator is not explicit in the *Ox* input specifications, usually it is not convenient to use attribute definitions for order-sensitive side effects such as code generation.

A common general approach to translation is to build and decorate a parse tree or syntax tree (meanwhile performing some of the checks for semantic errors), and to then make one or more determinate-order tree traversals for final error checks, gathering of compilation statistics, code generation, etc.

*Ox* has a facility for specification of such traversals, and this is the topic of section 1.9.

1.9 Postdecoration Traversals

The idea of decoration was described in section 1.7.3. *Postdecoration* refers to any time after the final decoration of the parse tree, which follows parsing of a correct input.
1.9.1 Example: Infix to Prefix Translation

The problem of parsing infix arithmetic expressions, and their translation to prefix form serves to introduce Ox’s postdecoration traversal facility.

The tokens of the example language are determined by the following L-file:

```
%{
#include "y.tab.h"
#include "ox.out.h"
%

%%

[ \n\t]*
[0-9]+        return(CONST); @ { sscanf(yytext,"%d",&@CONST.val@); @} \
(        return('(');
)        return(')');
+        return('+');
*        return('*');
.
    fprintf(stderr,"illegal character\n");
%%

The following Y-file completes the specification of the evaluator.

```

```
%token CONST
%left '+'
%left '*'

@attributes {int val;} CONST
@traversal @lefttoright @preorder LRpre

%
#include "ox.out.h"
#include <stdio.h>
%
```
The sequence: `@traversal @lefttoringh @preorder LRpre` specifies that a left-to-right preorder traversal of the parse tree be performed by the evaluator after the final decoration, and that the traversal be identified as `LRpre`. Note that `LRpre` is programmer-defined, and is not an Ox reserved word.

Each attribute reference section in the above Y-file contains a `traversal action specifier` starting with the `traversal mode annunciator` `@LRpre`, which is defined in the above-mentioned `@traversal` specification.

When the `LRpre` traversal reaches a node at which rule 1 is applied, an asterisk is printed, then each child of the node, in left-to-right order, is traversed. The behavior of the traversal at a node at which rule 2 is applied is the same, except that a plus sign is printed instead of an asterisk. When `LRpre` reaches a node for rule 3, no traversal action is performed, but the children of the node are traversed recursively as described above for nodes for rules 1 and 2. The `val` attribute of the `CONST` child is printed when a node for rule 4 is reached. When a traversal reaches a terminal node, no action is performed, and the traversal’s recursion degenerates.
1.9.2 General Description

Traversals Specifications

The Ox programmer may place in the declarations section (the part before the first `%%` mark) of the Y-file one or more traversals specifications. Such a specification consists of the reserved word @traversal, followed by a traversal specifier sequence and a non-empty sequence of identifiers, the identifiers being separated by whitespace. A traversal specifier sequence consists of the following (in any order):

- at most one of: @postorder, @preorder
- at most one of: @lefttoright, @righttolef t
- optionally: @disable

If neither @postorder nor @preorder appears in the sequence, the traversal is postorder by default. A left-to-right traversal is specified by default when neither @lefttoright nor @righttolef t appears.

Following the final decoration, the parse tree is traversed once for each traversal specification. The order of performing the traversals corresponds to the order of appearance of the traversal specifications. The @disable reserved word causes the generated evaluator to skip any traversal in whose specification it appears, which may be useful for debugging.

The code fragment:

```
@traversal @preorder LRpre
@traversal LRpost
```

appearing in the declarations section specifies that, after the final decoration, the generated evaluator is to perform a left-to-right preorder traversal named LRpre, followed by a left-to-right postorder traversal named LRpost.

Traversals Action Specifications

In addition to attribute definitions (section 1.5.2), the attribute reference sections of the Y-file may contain traversals action specifications. Each of these consists of a traversal mode annunciator, followed by a sequence of dynamic traversal modifiers and a traversal action. A traversal mode annunciator is @ followed immediately by the name of a previously-declared traversal.
Suppose traversal specifications of \texttt{LRpre} and \texttt{LRpost} as above. Then in the code fragment

\begin{verbatim}
s : expr
  @\{  @LRpost printf("\n");  /* 1 */
      @LRpost @revorder (1) printf("postfix: ");  /* 2 */
      @LRpre  @revorder (1) printf("\n");  /* 3 */
      @LRpre  printf("prefix: ");  /* 4 */
   @\}
;
\end{verbatim}

the attribute reference section has four traversal action specifications and no attribute definitions. Each specification is announced by either \texttt{@LRpre} or \texttt{@LRpost}. Each of the \texttt{printf} statements constitutes a traversal action. The form of a traversal action is that of a C code fragment, except that it may contain references to the attribute occurrences of the associated rule.

The second and third specifications each have \texttt{@revorder (1)} as a dynamic traversal modifier. A dynamic traversal modifier is either \texttt{@revorder} or \texttt{@reversedirection}, followed by a parenthesized expression that conforms to C syntax, except that it may refer to the rule’s attribute occurrences. \texttt{@revorder} and \texttt{@reversedirection} may each occur at most once in a given traversal action specification. If \texttt{@reversedirection} appears in two traversal action specifications within a given attribute reference section, the two specifications must have different annunciators. Dynamic traversal modifiers are used to override the traversal specifications of a given traversal when it reaches a given kind of node. The modifier \texttt{@revorder expr} means roughly “reverse order if expr”. When the \texttt{LRpre} traversal reaches a node at which the rule \texttt{s : expr} is applied, the expression \texttt{(1)} is evaluated, and because it is nonzero, the third traversal action, which prints a line feed, is executed as if \texttt{LRpre} were a postorder traversal, i.e., \textit{after} the recursive traversal of the subtree rooted at the node’s sole child. The execution of the fourth traversal action, \texttt{printf("prefix: ");} is not affected by any dynamic traversal modifier, and occurs according to \texttt{LRpre}’s (static) specification, i.e. \textit{before} the traversal of the child subtree.

When the \texttt{LRpost} traversal reaches a node at which \texttt{s : expr} is applied, the second traversal action is executed, the traversal proceeds to the child subtree, then the first traversal action is executed.
Traversals Semantics

The behavior of postdecoration traversals was illustrated in the preceding examples. In view of those examples, the following C-like pseudocode holds no surprises, but describes such behavior generally and precisely. The traversals are carried out by a single call of doTraversals (below) after the final decoration.

```c
enum orderType {PREORDER, POSTORDER};
enum directionType {LEFTTORIGHT, RIGHTTOLEFT};

enum orderType staticOrder(traversal T)
    {if (@preorder appears in the traversal specification of T)
       return PREORDER;
       return POSTORDER;
    }

enum directionType staticDirection(traversal T)
    {if (@righttoleft appears in the traversal specification of T)
       return RIGHTTOLEFT;
       return LEFTTORIGHT;
    }

int isDisabled(traversal T)
    {if (@disable appears in the traversal specification of T)
       return 1;
       return 0;
    }
```
void pdTrav(parse_tree_node N, traversal T)
{
    grammar_rule R;  /* the rule applied at N */
    enum orderType order[Z];  /* Z >= # of traversal action specs
                                for T in R */
    for T in R
    {
        enum directionType direction;
        int i,j,k;

        R = the grammar rule applied at N;
        let the traversal actions for T in the attribute definition section
          of R be numbered from 0 to k-1;
        for (i=0; i<k; i++)
        {
            if (the ith traversal action specifier has no @revorder)
                order[i] = staticOrder(T);
            else if ((the expression associated with @revorder) == 0)
                order[i] = staticOrder(T);
            else if (staticOrder(T) == POSTORDER)
                order[i] = PREORDER;
            else
                order[i] = POSTORDER;

            if (the ith traversal action specifier has no @revdirection)
                direction = staticDirection(T);
            else if ((the expression associated with @revdirection) == 0)
                direction = staticDirection(T);
            else if (staticDirection(T) == LEFTTOLeft)
                direction = RIGHToLEfT;
            else
                direction = LefTToLeft;

            for (i=0; i<k; i++)
            {
                if (order[i] = PREORDER)
                    execute the ith traversal action;
            }
        }
    }

    number the children of N from left to right
    with integers from 0 to j-1;
    if (direction == LEFTTOLeft)
        for (i=0; i<j; i++) pdTrav(the ith child of N,T);
    else
        for (i=j-1; i>=0; i--) pdTrav(the ith child of N,T);
    for (i=0; i<k; i++)
    {
        if (order[i] = POSTORDER)
            execute the ith traversal action;
    }
```c
void doTraversals()
{
    int i, k;
    parse_tree_node r;

    r = the root of the parse tree;
    k = the number of traversals;
    number the traversals from 0 to k-1, according to
    the order of appearance of their specifications;
    for (i=0; i<k; i++)
        if (!isDisabled(the ith traversal))
            pdTrav(r, the ith traversal);
}
```

### 1.10 Macros

Ox's input specification may be such that the same or similar text appears in more than one place in attribute reference sections. Ox has a macro substitution feature that can be used to decrease verbosity in such cases.

#### 1.10.1 Macro Definitions

Ox macros are defined in the declaration section of the Y-file. Such a definition consists of the `@macro` reserved word, an identifier (the name of the macro), a left parenthesis, a parameter list, a right parenthesis, the body of the macro, and the `@end` reserved word. The parameter list is a possibly empty sequence of identifiers, each (including the last, if the list is nonempty) followed by a comma. Each identifier is a sequence of letters and digits, beginning with a letter. The body of the macro is a segment of arbitrary text, terminated by the first occurrence of `@end`, with the following exceptions: When inside a comment or a string, or when preceded immediately by the backslash escape character, an occurrence of `@end` is considered part of the macro body (hence does not terminate the macro). Such a backslash character is deleted from the macro body.
1.10.2 Macro Uses

Ox macros are used only in attribute reference sections and in other Ox macros. Substitution occurs where a macro use is encountered outside of a string, comment, or attribute name.

A macro use consists of the name of a previously-defined macro, and an argument list in parentheses. The argument list is a possibly empty sequence of text fragments, each (including the last) such fragment terminated by a comma. In expanding a macro use, each text fragment is substituted for each occurrence in the macro body of the corresponding parameter in the macro definition. If commas, parentheses, or backslashes are to appear in a text fragment, they must be preceded by backslash escape characters, which are removed during substitution.

It is not necessary that the definition of a macro precede that of another macro in which it is used, as no macro substitution occurs until Ox processes the attribute reference sections.

1.10.3 Example

The following excerpts from a Y-file illustrate the use of Ox macros.

```y
@macro exprdefs(op,)
 @i @expr.env = @expr.env;
 @i @expr.2.env = @expr.env;
 @i @expr.type = typeResolve(@expr.1.type, @expr.2.type,);
 @i @expr.value = exprEval(op, @expr.type, @expr.1.type, @expr.2.type, @expr.1.value, @expr.2.value)
);
@end

@macro typeResolve(type1,type2,)
   ((type1 == type2)?type1:FLOATTYPE)
@end

```

```y
```

```
expr :   expr '/*' expr  
   @{ exprdefs('/*',) @}  
|   expr '/' expr  
   @{ exprdefs('/',) @}  
|   expr '+' expr  
   @{ exprdefs('+',) @}  
|   expr '-' expr  
   @{ exprdefs('-',) @}  
;  

The identifier exprEval referenced in the definition of the exprdefs macro is the name of either a C macro or C function. The Ox macro typeResolve above contains no Ox-specific constructs and, as a matter of style, could have been declared instead as a C macro or C function.

1.11 Shell Command Sequences for Evaluator Development

1.11.1 Conventions of Naming Ox Output Files

Ox translates the Y-file into a file destined for processing by Yacc, given the name ox.out.y. The L-files are translated into files destined for Lex. If there is exactly one L-file, its corresponding output file is named ox.out.1. If there is more than one L-file, the corresponding outputs are named ox.out.0.1, ox.out.1.1, ox.out.2.1, etc.

1.11.2 Review: Combining the Outputs of Yacc and Lex

In developing an ordinary (i.e., non-Ox) Yacc/Lex evaluator, y.tab.c and lex.yy.c can be compiled immediately into an executable file by placing the line

```
#include "lex.yy.c"
```

in a C-code section of the Yacc input specification [Lesk 75]. Alternatively, Yacc can be instructed (by using the -d command-line option) to produce a separate file y.tab.h that contains declarations needed by
both \texttt{y.tab.c} and \texttt{lex.yy.c}. The two files may then be compiled separately if the line

\begin{verbatim}
#include "y.tab.h"
\end{verbatim}

is placed in C-code sections of both the \texttt{Yacc} and the \texttt{Lex} input specifications. The two resulting object files can then be linked to produce an executable file.

### 1.11.3 Combined Use of Ox, Yacc, and Lex

There are certain declarations that must be visible from all of the files produced by \texttt{Ox}. By default, \texttt{Ox} produces files suitable for separate compilation, inasmuch as the \texttt{Yacc}-destined file and the \texttt{Lex}-destined file(s) each contain the common declarations. \texttt{Ox} also supports the one-step development approach described above. By placing \texttt{-h} on \texttt{Ox}'s command line, the designer calls for generation of a file \texttt{ox.out.h} containing the common declarations, which are then absent from \texttt{Ox}'s other output files. In this case, the line

\begin{verbatim}
#include "ox.out.h"
\end{verbatim}

is placed in the Y-file.

### 1.11.4 Typical Command Sequences

The following sequence of shell commands is an example of the separate compilation approach described. In this example, \texttt{Ox} translates the Y-file \texttt{ev.Y} into \texttt{ox.out.y} and the L-file \texttt{ev.L} into \texttt{ox.out.l}. The last command of the sequence links the two object files, yielding the executable file \texttt{ev}.

\begin{verbatim}
ox ev.Y ev.L
yacc -d ox.out.y
lex ox.out.l
cc -c y.tab.c
cc -c lex.yy.c
cc -o ev y.tab.o lex.yy.o -ll -ly
\end{verbatim}

The following command sequence does a one-step compilation.

\begin{verbatim}
ox -h ev.Y ev.L
yacc ox.out.y
lex ox.out.l
cc y.tab.c -ll -ly
\end{verbatim}
1.12 Other Points and Features

1.12.1 Error Recovery

Yacc has provisions for building parsers that attempt to recover from syntax errors, and the designer can use the words error, yyerrok, and yyclearin to implement such error recovery [Johnson 75]. When a parser that employs such techniques detects a syntax error, it may attempt to recover by popping items from its stack or by discarding tokens. During normal parsing, the Ox-generated evaluator synchronizes its stack operations with those of the Yacc-generated parser (see section 1.7). When the parser is built using error, yyerrok, or yyclearin, and a syntax error occurs, this synchronization is lost. It is possible for the evaluator to corrupt its stack and go out of control in such cases. Ox provides the function yyyerror to prevent such chaos. The parser calls yyyerror upon any syntax error, and the designer should write yyyerror such that yyyerror is executed at least once each time yyyerror is called. Any syntax error will then cancel parse-tree construction and attribute evaluation, and it is ensured that the Yacc-generated code can continue safely. Use of yyyerror is unnecessary but harmless if the Y-file makes no use of the words error, yyerrok, and yyclearin.

1.12.2 Memory Alignment

Many computing systems have hardware-related constraints on the addresses used for memory accesses. For example, for a certain type it may be required that the first byte of storage for each variable of that type reside at an even-numbered address. Then an instruction to access a variable of that type at an odd-numbered address results in a run-time error. When Ox is given the -aN command-line option, it produces an evaluator that aligns all C structures on addresses divisible by the integer \( N \). The default value for this alignment constant is 4, which is adequate for nearly all current computers.

1.12.3 Stripping Ox Constructs

Occasionally, the designer may wish copies of the Y-file and L-file(s) free of Ox-specific constructs. Suppose, for instance, that changes to the underlying grammar are under consideration, and that it is desired to test whether the
new grammar has parsing conflicts. To satisfy Ox semantics might require writing attribute definitions for any new rules. Ox’s output on ox.out.y could then be submitted to Yacc to test for parsing conflicts.

To avoid the above-mentioned writing of attribute definitions, the designer can use Ox’s -s command-line option, which filters all Ox-specific constructs from the inputs and yields files acceptable to Yacc and Lex. The original copies of the Y-file and L-file(s) are unchanged, but Ox’s outputs on ox.out.* contain neither Ox constructs nor the usual Ox-generated parse-tree management code.

1.12.4 Preventing Execution of Attribute Definition Code

Faulty user-written code in attribute reference sections may cause abnormal termination of the Ox-generated evaluator. For instance, dereferencing a stray pointer may corrupt the evaluator’s data structures and cause it to falsely report a cycle during attribute evaluation. The -n command-line option is a debugging feature that can be used to isolate the effects of anomalous attribute definition code. When Ox is used with this option, the generated evaluator uses the ready set as usual to determine an evaluation order for attribute instances, and still checks for cycles. Each time it is ready to solve an instance, however, it stops short of executing the code for the definition of that instance. When -n is used, the designer should take special notice of the effects upon other translation phases of such suppression of semantic analysis.

1.12.5 Control of Storage Allocation in the Generated Evaluator

When initializing itself, the Ox-generated evaluator allocates memory for its various data structures. When evaluating a large input, it may happen that the space allocated for a given data structure is inadequate. In such a case, the evaluator issues an error message indicating which data structure was exceeded and suggesting an appropriately larger size. The sizes of these data structures may be determined by the default values built into Ox, or by the evaluator designer’s use of the -Ya N option on the Ox command line, where
$a$ is an alphabetic character that specifies the data structure to be sized, and $N$ is an integer that determines the size of data structure $a$.

The evaluator designer can easily build an evaluator that accepts the same `-YaN` command-line options accepted by $\texttt{Ox}$. By specifying the `-YY` option on $\texttt{Ox}$'s command line, $\texttt{Ox}$ is instructed to declare in the generated evaluator a function $\texttt{yyyCheckForResizes}$ that can read $\texttt{main}$'s arguments (i.e., the command-line options passed to the generated evaluator) and adjust sizes accordingly. When using the `-YY` option, the designer should arrange the evaluator's $\texttt{main}$ program according to the following form:

```c
void yyyCheckForResizes();

main(argc, argv)
  int argc;
  char *argv[];
{
  
  /* This is executed before calling yyparse */
  yyyCheckForResizes(argc, argv);

  yyparse();

} /* main */
```

### 1.12.6 Parse Tree Statistics

Placing `-u` on $\texttt{Ox}$'s command line causes generation of an evaluator that prints, for each input, statistics regarding the parse tree built for the input. These include numbers of:

- terminal nodes and their attribute instances,
- nonterminal nodes and their attribute instances,

and other statistics.
1.12.7 Adjusting the Sizes of Ox’s Data Structures

Ox itself calls system memory allocation routines to obtain storage for its internal data structures. The default sizes of these data structures are quite generous, and exceeding them would be somewhat unusual. In case any of these is exceeded, Ox prints an error message indicating the use of a command-line option of the form `-XaN` to make $N$ the size of data structure $a$.

1.13 Example: An Integer Calculator

This section has Ox code for an evaluator of simple expressions involving multiplication and addition. Since the grammar has only synthesized attributes, the Ox implementation offers little advantage over one that uses only Yacc and Lex; it is presented as a very easy example of Ox usage.

The L-file specifies that the tokens are digit strings, parentheses, `'*'`, and ` '+'`

```c
[ 
\n\t\x]*   ;
[0-9]+    return(CONST); @{
    sscanf(yytext, "%d", &(CONST.val)); @}
\(        return('\(');
\)        return(')');
\+        return('+' );
\*        return(' *');
%%
```

```c
%
/* expr.L: L-file for a simple expression language */
#include "y.tab.h"
#include "ox.out.h"
%
```
The grammar is disambiguated by use of Yacc's %left reserved word. Each parse-tree node labeled by s, e, or CONST has an integer attribute instance named val. Use of the global variable sVal obviates postdecoration traversal.

/* expr.Y: Y-file for a simple expression language */
%left '+'
%left '*'
%token CONST

@attributes {long val;} s e CONST

{%
#include "ox.out.h"
long sVal;
%}

%%
s : e
0{ @i sVal = @s.val0 = @e.val0; 0}
;
  e : e '+' e
0{ @i @e.0.val0 = @e.1.val0 + @e.2.val0; 0}
;
  e : e '*' e
0{ @i @e.0.val0 = @e.1.val0 * @e.2.val0; 0}
  e : '(' e ')' 0{ @i @e.1.val0 = @e.1.val0; 0}
| CONST 0{ @i @e.1.val0 = @CONST.val0; 0}
;
%%

main()
{yyparse();
 printf("%d\n",sVal);
}
The following command sequence is used to build an executable file `calc` from the above specifications:

```
ox -h expr.Y expr.L
yacc -d ox.out.y
lex ox.out.l
cc -c y.tab.c
cc -c lex.yy.c
cc -o calc y.tab.o lex.yy.o -ly -ll
```

## 1.14 Example: A Binary Number Translator

This illustrates the use of Ox to build an evaluator based on an example attribute grammar that appears in the seminal paper on the subject [Knuth 68]. The input (after removal of whitespace) is either a nonempty string of binary digits or two such strings separated by a period. This input is interpreted as a binary representation of a floating point number, which is then printed on the standard output in its base-ten form.

Following is the text of the L-file:

```c
 %{ #include "y.tab.h"

    %
    [%]
        return ZERO;
    [%]
        return ONE;
    \.
        return DOT;
    [\n\t\v ]   ;
    .
        {fprintf(stderr,"illegal character\n");
           exit();
        }
```
Here is the text of the Y-file:

```yaml
%token ZERO ONE DOT

@attributes {float value; int scale;}
@attributes {float value; int scale,length;}
@attributes {float value;}

%start num

{%
#include <stdio.h>
float numValue;
%
```
%%

bit  :  ZERO
@{ @i @bit.value@ = 0;
    /* value is synthesized for bit. */
    /* scale is inherited for bit. */
}@;

bit  :  ONE
@{ @i @bit.value@ = twoToThe(@bit.scale@);
}@;

bitlist : bit
@{ @i @bitlist.value@ = @bit.value@;
    @i @bitlist.scale@ = @bitlist.scale@;
    @i @bitlist.length@ = 1;
    /* value and length are synthesized for bitlist. */
    /* scale is inherited for bitlist. */
}@;

| bitlist bit
@{ @i @bitlist.0.value@ = @bitlist.1.value@ + @bit.value@;
    @i @bitlist.0.scale@ = @bitlist.0.scale@;
    @i @bitlist.1.scale@ = @bitlist.0.scale@ + 1;
    @i @bitlist.0.length@ = @bitlist.1.length@ + 1;
}@;

num : bitlist
@{ @i numValue = @num.value@ = @bitlist.0.value@;
    @i @bitlist.scale@ = 0;
    /* value is synthesized for num. */
}@;

| bitlist DOT bitlist
@{ @i numValue = @num.value@ = @bitlist.0.value@ + @bitlist.1.value@;
    @i @bitlist.0.scale@ = 0;
    @i @bitlist.1.scale@ = -@bitlist.1.length@;
}@;

%%
main()
    {
        if (!yyparse())
            printf("%.30f\n", numValue);
        
    }

float twoToThe(in) /* returns 2 raised to the power in */
    int in;
    {
        if (in < 0) return (1.0 / twoToThe(-in));
        if (in == 0) return 1.0;
        else return (2.0 * twoToThe(in - 1));
    }

Job development of the above evaluator follows the separate compilation approach described in section 1.11.

Removing the Ox-specific constructs and the printf statement from the above source yields a pair of files that constitute a semantics-free recognizer of binary numbers.
1.15 Example: Translation from Infix to Postfix and Prefix

In this example, the generated evaluator is to perform two postdecoration traversals, one for printing the prefix form of a given infix expression, and one for printing the postfix form. The tokens of the language are specified as follows:

```c
#include "y.tab.h"
#include "ox.out.h"

char *lexeme()
{
    char *dum;
    dum = (char *)malloc(yylen+1);
    strcpy(dum,yytext);
    return dum;
}

%{
    /* L-file for translation of infix expressions */
    return(CONST); @{ @CONST.lexeme0 = lexeme(); @}
    return(ID); @{ @ID.lexeme0 = lexeme(); @}
    return('(');
    return(')');
    return('+');
    return('*');
    return('/');
    return('-');
  }

%}
```

```c
[ 
[0-9]*/
[A-Za-z_][A-Za-z_0-9]*
`\`
`\`
`\`
return(CONST);
`\`
return(ID);
return('(');
return(')');
return('+');
return('*');
return('/');
return('-');
```
The first traversal performed is named \texttt{LRpre}, and the second is named \texttt{LRpost}. By default, both are left-to-right traversals. \texttt{LRpost} is a postorder traversal by default. \texttt{LRpre} is specified as a preorder traversal.

/* Y-file for translation of infix expressions to prefix and postfix */
%token ID CONST
%start s
%left '+' '-'
%left '*' '/'

@attributes {char *lexeme;} ID CONST
@traversal @preorder LRpre
@traversal LRpost

{%
#include "ox.out.h"
#include <stdio.h>
%}
%%

s : expr
    @{ @LRpost printf("\n");
        @LRpost @reorder (1) printf("postfix: ");
        @LRpre @reorder (1) printf("\n");
        @LRpre printf("prefix: ");
    @}
expr : expr '/*/' expr
    @{ @LRpost printf(" * ");
        @LRpre printf(" * ");
    @}
    | expr '+' expr
    @{ @LRpre printf(" + ");
        @LRpost printf(" + ");
    @}
    | expr '/' expr
    @{ @LRpre printf(" / ");
        @LRpost printf(" / ");
    @}
    | expr '-' expr
    @{ @LRpre printf(" - ");
        @LRpost printf(" - ");
    @}
    | '(' expr ')'|
ID
    @{ @LRpost printf(" %s ",@ID.lexeme@);
        @LRpre printf(" %s ",@ID.lexeme@);
    @}
CONST
    @{ @LRpost printf(" %s ",@CONST.lexeme@);
        @LRpre printf(" %s ",@CONST.lexeme@);
    @}

; %

main()
{yyparse();
}
Chapter 2

General Design Criteria and Comparison with Other Systems

2.1 Generalization of the Function of Yacc

The concept of attribute grammar is a generalization of the concept of context-free grammar. The language understood by Yacc can express context-free grammars, so it is natural to extend that language so that it can express attribute grammars. Yacc is a very popular tool, and has been used to build evaluators for many languages. The above-mentioned extension of the language of Yacc permits augmentation of the semantics of any Yacc-implemented evaluator, without interfering with the implementation of the existing semantics.

Ox is designed to accept with little or no modification source code for any evaluator written for Yacc and Lex. Moreover, Ox is made to understand a more general language, one capable of expressing attribute grammars.

Ox inherits all of the familiar syntax and semantics of Yacc and Lex, so the Ox user is assured that the scanning and parsing behaviors of an Ox-generated evaluator follows those of the corresponding “pure Yacc/Lex” evaluator (i.e., the one gotten by deleting from Ox’s input all of the Ox-specific entities).
2.2 Generality of the Class of Attribute Grammars Accepted

For the sake of efficiency, many existing attribute-grammar evaluator generators accept only certain classes of attribute grammars, each such class being characterized by a particular pattern of attribute dependencies [Lorho 88]. While efficiency of the Ox-generated evaluators was a primary consideration, it was not permitted to restrict Ox’s generality. It was considered besides that gains from such restrictions would be marginal. The only such restriction on Ox’s input is that synthesized attributes of tokens may not depend on other attributes.

The underlying context-free grammar must be acceptable to Yacc: after applying Yacc’s disambiguating rules, the grammar must be LALR(1).

2.3 Ox as a Preprocessor

One approach to implementing Ox would be to modify Yacc and Lex themselves, extending them to perform the functions of Ox. There are various versions of Yacc and Lex, as well as workalikes such as Bison and Flex. Building Ox as a separate preprocessor facilitates its adaption for use with those various programs and obviates their modification.

2.4 Portability

To enhance the portability of Ox itself, the only programs required for its installation and use are Yacc, Lex, and a C compiler. Assuming the availability of an ordinary quasi-ANSI C compiler, the Ox implementation is free of compiler and hardware dependencies.

Ox is made to be a cross- evaluator-generator: it can generate code that runs on a machine other than the machine on which Ox itself runs. Moreover, for any given act of Ox-generating an evaluator, the choice of a sufficiently large alignment size (see section 1.12.2) ensures that the Ox-generated code is portable to almost every machine that runs a quasi-ANSI C compiler.
2.5 Competing Systems

By 1988, there had been implemented approximately forty software systems for generating evaluators from attribute grammars. These vary in many respects and are catalogued in [Lorho 88]. For each tool, there is some class of context-free grammars, typically LL(1), SLR(1), LALR(1), or LR(1), that underlies the class of attribute grammars accepted. Further, most of the tools have constraints (beyond that of non-circularity) on the attribute dependencies of the grammar. Many of the tools have a generality roughly equivalent to that of Ox, whereas many others are much less general. There is also wide variety with respect to the languages by which the attribute grammars are expressed, the languages in which the generated evaluators are written, and the algorithms used.

2.5.1 The Delft System

Perhaps Ox’s most important feature is its generalization of the languages of Yacc and Lex. The “Delft system” [van Katwijk 83] is the only Yacc preprocessor described in [Lorho 88]. It has been used in the development of several compilers, but has some notable shortcomings. The Delft system was developed on a PDP-11, at a time when computer main memory was relatively scarce. This is reflected in the generated evaluators, each of which, rather than building a parse tree, keeps attribute instances on a stack that runs parallel to the parser stack. This limits the system to attribute grammars whose parse trees can be decorated in a single traversal [Lorho 88]. The Delft preprocessor sometimes inserts into the Yacc rules new nonterminals that derive the empty string, and this may destroy the LALR(1) property of the grammar.

2.6 Performance

Since the evaluator builds a parse tree in main memory, memory size tends to limit the size of the input to be evaluated. Much consideration is given to efficient parse-tree storage techniques. In some cases, storage for subtrees whose attribute values are no longer needed is reclaimed.

The time efficiency of the generated evaluator was also a very important
design consideration. It is intended that any evaluator generated using Ox be competitive with the fastest equivalent evaluator that could be generated using Yacc, Lex, and hand-coded semantics. Part of the topological sort is performed at evaluator generation time, thus saving work for the evaluator.

Ability to compile a 10,000 line imperative-language program on a contemporary workstation in a normal amount of time was set as a rough standard of performance for the generated evaluators.
Chapter 3

Design of the Ox-generated Evaluator

This chapter presents an overview of the data structures and algorithms of the Ox-generated evaluators. Detailed understanding may be obtained by a concomitant perusal of actual Ox-generated code.

3.1 Topological Sort Using Dependee Counts

The order in which the attribute instances of a parse tree are solved is constrained by the dependency relations of the attribute grammar. That order is determined partly by Ox (at evaluator-generation time) and partly by the Ox-generated evaluator at decoration time. The algorithms used to determine the order are based on the topological sort algorithm presented in [Knuth].

For each attribute instance, there is a set of instances that depend (cf. page 11) upon that instance. The main idea is to maintain for each attribute instance a count of the not-yet-solved attribute instances upon which the given instance depends. Such a count is called a dependee count. When an instance \( i \) is solved, the dependee count of each instance that depends on \( i \) is decremented. Whenever such a decrementation causes the dependee count of an instance to reach zero, that instance is placed in a set of instances that are ready to be solved.

In the attribute definition section of each rule of an Ox input specifica-
tion, there appear definitions of the synthesized attribute occurrences of the LHS and of the inherited occurrences of the RHS. The occurrences so defined are called the output occurrences in the rule. An attribute instance corresponding to an output occurrence in a given rule is said to be output relative to that rule. Each output occurrence is defined in terms of zero or more other occurrences in the rule. For each output occurrence in a given rule, there is a semantic action, a code fragment whose execution solves instances corresponding to that occurrence. Such semantic actions for the synthesized attributes of tokens are translated into C code, placed directly in Lex actions, and executed at lookahead time. All other attribute occurrences are defined in the Y-file, and their semantic actions are translated into C and either:

1. placed directly in the rule's Yacc action and executed at reduction time, or

2. placed in a global table of semantic actions and executed at decoration time.

### 3.2 Static Phase of the Topological Sort

The tasks of:

- finding the attribute occurrences whose semantic actions can be placed directly in Yacc actions, and
- determining the order of placement of those semantic actions

are outlined in the psuedocode routine `staticTopSort` below. This routine also generates several tables needed by the generated evaluator. The use of these tables is described in section 3.6. Although the work of `staticTopSort` is done by Ox, its relation to the design of the generated evaluator makes appropriate its description in this chapter.
void staticTopSort(R:rule)    /* R is a rule in the Y-file */
{
    /* Z is an integer constant such that for any rule R,
        Z is larger than the number of attribute occurrences in R.
        The attribute occurrences in R are referred to by
        consecutive nonnegative integers.
    */
    setOfOccurrences LRS;     /* the local ready set */
    semAction SA[Z];          /* the array of semantic actions */
    integer RC[Z];            /* the array of dependee counts */
    setOfOccurrences DS[Z];   /* DS[o] is the set of dependents of o */
    /* RC[o] and SA[o] are unused if o is not an output occurrence */

    make LRS empty;            
    for each o an attribute occurrence in R
        make DS[o] empty;
    for each o an attribute occurrence in R
    {
        place the semantic action for o in SA[o];
        RC[o] = 0;
        for each p an attribute occurrence such that o depends upon p
        {
            RC[o] = RC[o] + 1;
            make o a member of DS[p];
        }
        if (RC[o] == 0)
            make o a member of LRS;
    }

    for each o an attribute occurrence in R
    { if (o is a synthesized occurrence in a token)
        make o a member of LRS;
    }

    while (LRS is not empty)
    {
        remove a member o from LRS;
        if (o is an output occurrence in R)
            append the C-translation of SA[o] to R’s Yacc action;
        for each p in DS[o]
        {
            RC[p] = RC[p] - 1;
            if (RC[p] == 0) make p a member of LRS;
        }
    }
}
for each o an attribute occurrence in R
   {if (o is not an output occurrence in R)
      {copy DS[o] into the table of lists of dependents
       in the generated evaluator;
      }
   else if (RC[o] != 0)
      {place the C-translation of SA[o] in
       the table of decoration-time-executed semantic actions
       in the generated evaluator;
       copy DS[o] into the table of lists of dependents
       in the generated evaluator;
      }
   }
for each o an output occurrence in R
   copy RC[o] into the dependee-count-initialization list for R
   in the generated evaluator;
}

3.3 Parse-Tree Nodes

The process of building a parse tree is described in section 1.7.1. The implementation is such that, given a node, the evaluator can determine easily:

- whether the node is a terminal or nonterminal node
- a serial number that identifies the rule applied at the node (if the node is a nonterminal node)
- the beginning of the list of the node’s children
- the node’s parent (if its parent has yet been created)
- the node’s location on the stack (if its parent has not yet been created)
- the storage area for the node’s attribute instances
- the beginning of the list of dependee counts for the node’s attribute instances
3.4 Items on the Parse-Tree Stack

The evaluator maintains a stack for building the parse tree (see section 1.7.1). Each item on the stack has three components:

- a pointer to the root of a subtree of the complete parse tree. Such a root is said to be a stacked node and such a subtree is a stacked subtree. A stacked node or stacked subtree is said to be on the stack.

- a list of the solved synthesized instances in the root of the stack item’s subtree, if the root is a nonterminal. The use of this list is describe in section 3.7.

- a pointer to the oldest node in the stack item’s subtree. This component is used for memory management, and its use is described in section 3.11.1.

3.5 Addressing Attribute Instances and Dependeec Counts

For a given parse-tree node and a given attribute occurrence in the rule applied at the node, there is an attribute instance and a dependee count. The generated evaluator can address an attribute instance or its dependee count given a node and:

a symbol position: an integer that indicates the position within the rule of the grammar symbol of the attribute occurrence. This is used to find the node to which the instance belongs. If the occurrence is on the rule’s LHS, the instance belongs to the given node. Otherwise the instance belongs to one of the node’s children, and the symbol position determines an offset from the beginning of the list of the node’s children.

an attribute number. Having found a node using a symbol position, the attribute number is used to specify a certain attribute instance in that node. The attribute number is used as an offset from the beginning of the node’s list of dependee counts.
3.6 Static Data Structures of the Generated Evaluator

The tables generated statically as indicated in the pseudocode routine `staticTopSort` (section 3.2) are copied into the generated evaluator. This section describes their use.

3.6.1 Table of Dependeecount-Initialization Lists

Each shift or reduction corresponds to an application of a certain rule of the attribute grammar (see section 1.7.1). When such a stack operation is performed, the generated evaluator initializes the dependee counts of the instances that are output relative to the rule being applied. Those dependee counts are initialized with the values calculated and stored in `staticTopSort`'s array RC. The generated evaluator has such a list for each rule. Each item on such a list is an ordered triple referring to an attribute occurrence in the rule being applied, and contains a symbol position, an attribute number, and the value with which the dependee count is to be initialized.

3.6.2 Table of Decoration-time-executed Code Fragments

The instances not solved as part of a Lex or Yacc action are solved at decoration time. The semantic actions whose executions are required for solving those instances are stored in a global table indexed by a triple consisting of a rule number, a symbol position, and an attribute number.

3.6.3 Table of Lists of Dependents

As mentioned above (section 1.5.1), there are, for each parse-tree node, two rules of interest: the home rule and the parent rule. The home rule and parent rule of a given attribute instance are the same as the home rule and parent rule of the node to which the instance belongs. A given instance corresponds to exactly two attribute occurrences: one relative to its home rule and one relative to its parent rule. Thus, a given attribute instance
may have dependents relative to its home rule and dependents relative to its parent rule.

To find the list of dependents of an attribute occurrence relative to a given rule, the table of dependent lists is indexed by the rule number, symbol position, and attribute number of the occurrence.

Decrementing a dependee count of a dependent instance is part of signaling that dependent instance. The other part of the signalling operation is to place the signalled instance in the ready set if its dependee count is being decremented to zero. It may be said that the dependee signals the dependent.

3.7 Solved Instances Belonging to Parentless Nonterminal Nodes

Stacked nodes do not yet have parents. It is possible (due to execution of a Yacc action, for example) that some of the synthesized attribute instances of a stacked nonterminal node have been solved. Clearly it is not immediately possible to signal the dependents relative to the parent rule of any instances so solved. When an attribute instance belonging to a stacked nonterminal node is solved, the attribute number of that instance is placed on a list associated with the node’s position on the stack (see section 3.4). When the node’s parent is created (during a reduction), the dependents (relative to the parent rule) of the node’s solved synthesized attributes (as remembered on the list) are signalled, and the list is made empty.

3.8 Operations on the Parse-Tree Stack

As indicated in section 1.7.1, the evaluator keeps a stack for the purpose of building the parse tree, and operations on it parallel the parser’s operations on its own stack.

3.8.1 Synchronization of Stacks

Ox places code inside Yacc and Lex actions, so that control passes to Ox-generated code during execution of each such action. Thus it is easy to make
the evaluator’s reduction and lookahead operations coincide with those of
the parser. Ox-generated code does not obtain control during the parser’s
shift operations; parser shifts can only be detected after they have occurred—
by examining the parser’s data structures. Reading the parser’s lookahead
buffer \texttt{yychar} [Johnson 75] is a simple and portable way for the evaluator
to determine whether it needs to shift before proceeding with the impending
reduction or lookahead. The value in \texttt{yychar} is positive exactly when the
parser has obtained a token by lookahead but has not yet performed the shift
corresponding to that lookahead.

3.8.2 Buffering of Images of New Terminal Nodes: Lookahead

If the evaluator’s lookahead buffer is not empty when the evaluator obtains
control from inside a \texttt{Lex} action, the evaluator performs a shift in order
to empty the buffer. The semantic actions for the synthesized attribute
instances of the new node are executed. Those instances are thus solved and
their values stored in the evaluator’s lookahead buffer.

3.8.3 Creation of Terminal Nodes: Shifting

Allocation of memory for a new terminal node is the first step in the eval-
uator’s shift operation. The values in the evaluator’s lookahead buffer are
copied into that memory, and the dependee counts of the node’s synthesized
instances are initialized to zero. A pointer to the new node is pushed onto
the parse-tree stack, and the new node is marked as parentless and childless.
The new node is recorded as the oldest node (see sections 3.4 and 3.11.1) in
the new subtree.

3.8.4 Creation of Nonterminal Nodes: Reduction

The following sequence is carried out during each execution of a \texttt{Yacc}
action. \(R\) is the rule being applied in the reduction and \(k\) is the length of \(R\)’s RHS.

1. Execute any user-supplied \texttt{C} code in the action (see section 1.7.2).

2. If the parser’s lookahead buffer is empty and the evaluator’s lookahead
buffer is not empty, perform a shift.
3. Allocate memory for the new node.

4. Make the $k$ nodes on top of the stack the children of the new node.

5. Make the new node the parent of each of the $k$ nodes on top of the stack.

6. Label the new node with the serial number of $R$.

7. Number the child nodes 1 to $k$ from left to right.

8. Execute any semantic actions placed in the action by Ox during the static phase of the topological sort (see section 3.2).

9. Traverse the dependee-count-initialization list for $R$ and initialize the dependee counts of the output instances (relative to $R$) according to the list.

10. For each child node, signal the dependents (relative to $R$) of the instances on the list of solved synthesized attribute instances (see section 3.7) of the node.

11. For each attribute instance $i$ whose dependee count has just been initialized to zero:

   (a) if $i$ belongs to the new node, place $i$ on the list of solved synthesized attribute instances of the new node.

   (b) if $i$ belongs to a nonterminal child $c$ of the new node, signal the dependents of $i$ relative to the rule applied at $c$.

12. If $k$ is zero, record the new node as the oldest node (see sections 3.4 and 3.11.1) in the new subtree.

13. Push the new subtree onto the stack.

### 3.9 Dynamic Phases of the Topological Sort

#### 3.9.1 Decoration

The ideas of decoration and the ready set were introduced in section 1.7.3.
Decorations are performed periodically to avoid exceeding the size limit of the ready set. The evaluator maintains a counter that is decremented each time a nonterminal node is created. When the counter reaches zero, it is set to a certain implementation-dependent positive value, and a decoration is performed. The final decoration occurs after parse-tree construction is completed.

For each decoration, the following sequence is repeated until the ready set is empty:

1. Remove from the ready set an attribute instance \( i \) belonging to a node \( n \).
2. Execute the semantic action for \( i \) (see section 3.6.2).
3. If \( i \) is an inherited instance:
   (a) If \( n \) is a nonterminal node, signal \( i \)'s dependents relative to its home rule.
   (b) signal \( i \)'s dependents relative to its parent rule;
4. If \( i \) is a synthesized instance:
   (a) signal \( i \)'s dependents relative to its home rule.
   (b) If \( i \) has a parent, signal its dependents relative to its parent rule.
   (c) If \( i \) is parentless, place \( i \) on the list of solved synthesized instances (see sections 3.4 and 3.7) for the subtree rooted at \( n \).

Recall (section 3.6.3) that signalling may add instances to the ready set. Thus the process of decoration may cause the ready set to grow and shrink before its emptying results in termination of the decoration.

### 3.10 Stack Implementation of the Ready Set

The topological sort algorithm presented in [Knuth] specifies the use of a ready queue rather than a ready set. Use of a queue is not an essential feature of the algorithm, and the Ox-generated evaluator implements the ready set as a stack.
For each parse tree, there may be envisioned a *dependency graph* [Aho 86]: a directed graph whose nodes are the attribute instances in the tree, and whose edges are directed from dependees to dependents. Implementing the ready set as a queue effects a breadth-first traversal of the dependency graph, whereas a stack implementation effects a depth-first traversal.

The parse trees (hence the dependency graphs) for long programs tend to be *broad*, while those for programs with deep nesting tend to be *deep*. There are practical limits on the depth of nesting of human-readable programs, but program length is far more variable. The stack implementation permits choosing for the ready set a size that is exceeded only for programs that are nested very deeply. In view of the evaluator’s arrangement of parse-tree storage (section 3.11.1), the depth-first behavior of the sort also improves the evaluator’s locality of reference, which may reduce the frequencies of cache misses and page faults.

### 3.11 Storage Management Scheme

#### 3.11.1 Parse-Tree Storage

At the time of initialization of the generated evaluator, `malloc` is used to obtain a large amount of memory dedicated to storage of parse-tree nodes. Such memory is subsequently allocated one node at a time (upon each shift and each reduction) by code internal to the evaluator.

The evaluator maintains a pointer to the beginning of the available space for node storage. When a node is allocated, that pointer is incremented by the size of the allocated node. For parse-tree nodes `n` and `m`, the pointer to `n` is smaller than the pointer to `m` if and only if `n`’s creation preceded the creation of `m`. Further, the node storage space is used *exclusively* for node storage: if `d` is a datum whose address is greater than that of `n` and less than that of `m`, then `d` is a parse-tree node.

When a node is created during a shift or during an epsilon reduction, its address is written to the oldest-node component of the topmost stack item (see sections 3.4, 3.8.3, and 3.8.4).

During each reduction that is not an epsilon reduction, a new subtree is built from one or more stacked subtrees. The root of the new subtree occupies the same position on the stack as did the oldest (leftmost) one of
the subtree(s) from which it is built. The oldest node of the new subtree is the same as the oldest node of the tree formerly occupying the new subtree’s stack position. Thus the oldest-node component of the topmost stack item need not be reassigned in order to preserve the following invariant:

Between stack operations, the subtree on top of the stack occupies exclusively the space from the location indicated by the oldest-node component of the top stack item up to but not including the beginning of the available node-storage space.

3.11.2 Dependee-Count Storage

The evaluator maintains a space exclusively for dependee counts. Storage for dependee counts is allocated in parallel with allocation of parse-tree nodes. Each parse-tree node has a pointer to the beginning of the list of dependee counts for the node’s attribute instances. Each such list is allocated at the time of creation of the node to which it belongs. Thus the dependee counts of the subtree on top of the stack occupy exclusively a contiguous segment of memory from the beginning of the oldest node’s dependee-count list up to but not including the beginning of the available dependee-count-storage space. That segment is all zeroes exactly when all of the attribute instances in the subtree have been solved. A very efficient way of detecting unsolved instances in a given subtree is obvious. The special case of checking for unsolved instances in the completed parse tree after the final decoration is equivalent to checking the parse tree for cycles.

3.11.3 Pruning

To prune a subtree is to remove all of its descendents and to reclaim all storage (nodes, instances, dependee counts) associated with those descendents, while preserving the storage of the subtree’s root.

The preceding discussions of storage organization suggest an efficient algorithm for pruning the subtree on top of the stack:

1. The pointer to the beginning of the available node-storage space is assigned the oldest-node component of the top stack item.
2. New space is allocated for storing the root of the pruned subtree, i.e.,
the pointer mentioned in the previous step is incremented by the size
of the root node.

3. The root node is copied to its new place in memory.

4. The root-pointer component (section 3.4) of the top stack item is made
to point to the new location of the root node.

5. The storage for dependee counts of the reclaimed nodes is reclaimed in
a manner analogous to that of the preceding steps.

The cost of the above pruning procedure is independent of the size of the
pruned subtree, and depends only on the size of its root.

Clearly there are cases in which pruning would destroy needed infor-
mation. Section 4.4.2 discusses static evaluation of conditions sufficient for
pruning.

3.11.4 Example: Stack Operations and Pruning
Suppose the Y-file contains the rules:

\[ A : \text{B c D \{ \ldots \} } ; \]
and

\[ D : \text{\{ \ldots \} } ; \]

where \( A, B, \) and \( D \) are nonterminals and \( c \) is a token. The contents of the
attribute reference sections of the given rules are irrelevant to the discussion.
Suppose that \( B \) derives a token string whose first symbol is \( \text{a} \). In the diagrams
that follow, the stack pointer is shown as \( \text{stackTop} \), and the pointer to the
next available space for parse-tree storage is shown as \( \text{freeSpace} \). Each item
shown on the stack is a pair consisting of (on the right) a pointer to the root of
a subtree, and (on the left) a pointer to the oldest node in that subtree. The
lists of solved synthesized instances are not shown (cf. section 3.4). Details
of parse tree nodes, such as parent pointers, child pointers, and attribute
instances are also omitted.

The following diagram shows a case wherein a subtree whose root is la-
beled \( B \) is on top of the stack (at position \( i \)).
The yield of that subtree is the token string hypothesized above. The node labeled \( a \) is thus the oldest descendant of the node labeled \( B \), which is the newest of all nodes.

Now consider the operation of shifting a node labeled \( c \). The new node is both the root and the oldest node of the subtree now on top:

An epsilon (empty RHS) reduction is quite like a shift:
The next diagram shows a reduction involving a rule with a nonempty RHS. Space has been allocated for a new node labeled A, which is the root of the new subtree indicated by stack item i (now the top of the stack). Note that the oldest-node component of stack item i is unchanged.

Following is the result of pruning the subtree just created. Storage for all of that subtree’s nodes has been reclaimed, and the node labeled A has been recreated in the place formerly occupied by its oldest descendent.
Chapter 4

Design and Implementation of the Evaluator Generator

This chapter presents some of the less obvious features and problems involved in the construction of Ox.

4.1 Size of Source Code

Ox has been implemented in about 200 kilobytes of code classified as follows:

<table>
<thead>
<tr>
<th>input to</th>
<th>approximate number of:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>files</td>
</tr>
<tr>
<td>Lex</td>
<td>6</td>
</tr>
<tr>
<td>Yacc</td>
<td>1</td>
</tr>
<tr>
<td>C compiler</td>
<td>14</td>
</tr>
</tbody>
</table>

4.2 Lexical Analysis

4.2.1 Limits of Lex

One design requirement (section 2.4) was that Ox’s installation require only Yacc, Lex, and C. Ox’s early development was done on a Hewlett-Packard system running a derivative of Unix System V. Lex was used to generate a single function to perform all of Ox’s scanning. The Lex specification for
that scanner was rather large, requiring the use of command-line options to expand some of Lex's internal data structures. When porting Ox to systems based on BSD Unix 4.3, Lex's data structures were again exceeded, but those versions apparently had no mechanism for expanding said data structures.

4.2.2 Use of Several Scanners

To avoid the above-mentioned limitations in some versions of Lex, the specifications for scanners must be kept small. Thus the task of scanning Ox input is partitioned into phases, each phase being performed by a distinct scanner. For each scanner, there is a Lex specification file, from which is generated a lex.yy.c file. Each such file contains a yylex function, which must be renamed so that the scanners may be distinguished. The several scanners so produced are called from a handwritten yylex consisting of a single switch statement. The switch selects the appropriate scanner according to the value of a context-dependent global variable.

Merely renaming the yylex functions generated from the several source files is not sufficient to avoid name conflicts, since each lex.yy.c file contains many global variables whose names are the same in each file. Those global variables all have the yy prefix, so it is easy to change their names by editing Lex's outputs mechanically. This is done by using Lex to generate a mechanical editor that reads a character from its command line and inserts that character after each yy encountered in its input. The edited files are then compiled separately and linked with Ox's other object modules.

Ox has a distinct scanner for each of the following segments of its inputs:

1. the declarations section (the part preceding the first \% mark) of the Y-file.
2. the Y-file from the first \% mark to end of the Y-file.
3. the L-file.

A fourth Lex specification is dedicated to the task of filtering out Ox constructs when using the \-S command-line option (see section 1.12.3).
4.2.3 Macro Processing

Macro definitions, which appear in the declarations section of the Y-file, are copied into Ox's memory, whence they are interpreted when a macro use is encountered. When a macro is not being interpreted, the scanner is receiving input from the Y-file or the L-file. If it then recognizes an identifier in the context of an attribute reference section, the scanner searches the set of macro definitions for a macro named by that identifier. If such is found, Ox's macro interpreter becomes the source of the scanner's input.

The main job of the macro interpreter is to maintain a pointer that indicates either:

- a character in Ox's copy of the macro definition, or
- a character in a copy of an actual parameter to the macro use.

The indicated character is returned to the scanner whenever the scanner's character-input function is called.

Macros may use other macros. Thus a macro use may be encountered while the scanner is getting its input from the macro interpreter. In this case the context of the current interpretation is saved on a stack belonging to the macro interpreter, a new context is created, and interpretation of the nested use begins. When the interpreter's character pointer reaches the end of a copy of a macro definition, it checks its stack to see if the just-finished interpretation was nested. If so, interpretation of the enclosing context resumes and the scanner continues to receive input from the macro interpreter. If the stack is empty, the scanner resumes its input from the Y-file or L-file.

4.3 Syntax Analysis

Yacc is used to generate Ox's parser. The grammar is an extension of the Yacc grammar for Yacc given in [Johnson 75]. The latter grammar has 33 rules, while the grammar for Ox has about 110 rules.

4.4 Semantic Analysis

The present implementation analyzes Ox rules one at a time: relations of rules to one another (such as might reveal a circular grammar, for instance)
are not considered, except for the consistency and completeness checks mentioned in section 1.5.1.

4.4.1 Topological Sort

Because of its relevance to the design of the Ox-generated evaluator, the static phase of topological sorting is described elsewhere (section 3.2).

4.4.2 Pruning Conditions

Consider the situation wherein the Ox-generated evaluator has just applied a grammar rule $R$ and then performed a decoration. At such time, the ready set is empty, and there is on top of the stack a subtree whose root is labeled by $R$. The present section considers the problem of deciding statically whether in such a situation the new subtree may be pruned safely (recall the pruning algorithm from section 3.11.3).

The static pruning test must ensure that despite pruning, all attribute instances in the parse tree are solved correctly and that they are available for reference during traversals. The latter requirement precludes pruning in evaluators that perform postdecoration traversals.

Let $o$ be an attribute occurrence in a rule $R$. Then $o$'s dependee closure in $R$ is the set of occurrences computed by the following algorithm:

\[
\begin{align*}
&\text{temp} \leftarrow \{\}\; ; \\
&\text{DC} \leftarrow \{o\}; \\
&\textbf{while} \text{ temp} \neq \text{DC} \\
&\hspace{1cm} \textbf{begin} \\
&\hspace{1cm} \text{temp} \leftarrow \text{DC}; \\
&\hspace{1cm} \textbf{foreach} p \text{ in temp} \\
&\hspace{1cm} \hspace{1cm} \textbf{if} p \text{ depends upon an occurrence } q \text{ in } R \\
&\hspace{1cm} \hspace{1cm} \text{then} \text{ DC} \leftarrow \text{ DC } \cup \{q\}; \\
&\hspace{1cm} \textbf{end} \\
&\textbf{end}
\end{align*}
\]

Upon termination, $DC$ holds the dependee closure of $o$.

Note that unless $o$ is an output occurrence in $R$, $o$'s dependee closure in $R$ is $\{o\}$.

The following analysis assumes:
• A decoration is performed immediately after the reduction.

• The grammar is noncircular. ¹

• There are no postdecoration traversals.

• In order to solve a given instance, all instances upon which it depends must have already been solved and must still exist at the time the given instance is solved.

The Pruning Conditions C1 and C2  The following conditions on a rule $R$ are necessary and sufficient for safe pruning after reductions involving $R$:

C1 If $o$ is a RHS inherited occurrence in $R$, $o$'s dependee closure in $R$ contains no LHS inherited occurrence.

C2 If $o$ is a LHS synthesized occurrence in $R$, at least one of the following is true:

- C2a $o$'s dependee closure in $R$ contains no RHS occurrence.
- C2b $o$'s dependee closure in $R$ contains no LHS inherited occurrence.

Note that the condition on LHS output occurrences (C2) is milder than the condition on RHS output occurrences (C1).

C1 and C2 are Sufficient for Safe Pruning  Condition C1 together with noncircularity ensures that all non-root instances in the subtree are solved no later than the post-reduction decoration: The grammar is noncircular, so for any nonroot instance $i$ unsolved after the post-reduction decoration, there remains a chain of dependencies that includes both $i$ and some inherited instance in the root. Existence of such a chain is precluded by C1.

Now consider the solving of the synthesized instances in the root of the new subtree. Suppose $R$ satisfies C1 and C2, and let $i$ be a synthesized instance in the root. If C2a is true for the occurrence corresponding to $i$, then the attribute instances in the RHS are not needed to solve $i$. If C2a fails, then C2b holds, and $i$ is solved no later than the post-reduction decoration.

¹Because the present implementation assumes noncircularity rather than testing for it, the subtree to be pruned must be checked for unsolved instances prior to pruning. Such evaluation-time tests are quite inexpensive, as indicated in section 3.11.2.
**C1 and C2 are Necessary for Safe Pruning** Suppose that C1 is false. Then there is a RHS inherited occurrence $o$ whose dependee closure contains a LHS inherited occurrence $p$. The instance corresponding to $p$ remains unsolved immediately after the post-reduction decoration, since the parent rule (section 1.5.1) of that instance is not yet known. It follows that the instance corresponding to $o$ also remains unsolved at that time. Pruning for $R$ must be forbidden, since it would remove an unsolved instance.

Suppose that C2 is false. Then there is a LHS synthesized occurrence $o$ whose dependee closure contains a RHS occurrence $p$ and a LHS inherited occurrence $q$. The instance corresponding to $q$ cannot be solved prior to pruning, since its parent rule is unknown. Thus the instance corresponding to $o$ must be solved after the pruning in question. But the solving of that instance must precede the pruning, since one of its dependents, namely the instance corresponding to $p$, is to be removed during pruning. Thus pruning is forbidden when C2 fails.

### 4.5 Code Generation

A copy of the grammar-independent part of the Ox-generated code is stored in Ox itself, so that execution of Ox does not depend on access to a library.

The grammar-dependent tables and semantic actions are placed in about ten temporary files that are later assembled and placed on ox.out.*.

### 4.6 Validation of Ox

The evaluators mentioned in sections 5.1 and 5.2 are the main parts of a suite of programs used in the automated testing of Ox.

For each evaluator in the test suite, there are stored correct copies of ox.out.y and ox.out.1. If the test version of Ox generates files that differ from these correct copies, the differences are reported, and a new executable evaluator is built using the newly-generated copies of ox.out.y and ox.out.1. For each evaluator, there is stored a set of test input files, and for each such input, a file containing the correct corresponding output. The outputs of the new evaluator are compared to the correct outputs, and any differences are reported.
Chapter 5

Experience with Ox

5.1 Small Languages

Several toy evaluators have been built using Ox, including:

- a binary to decimal floating-point number translator (see section 1.14),
- an infix to postfix-and-prefix expression translator (see section 1.15),
- a calculator for arithmetic expressions,
- a program that prints the parse trees of expressions

5.2 GPPL: A Block-structured Imperative Programming Language

Construction of a compiler for a language called GPPL has been the most significant Ox programming experience.

5.2.1 General Description

GPPL is described roughly as a block-structured quasi-subset of C. GPPL’s syntax is specified in about 80 Yacc rules. A more detailed description of GPPL and its implementation is given in [Bischoff 92].
Types

GPPL has integer, floating point, boolean, and void types, but lacks compound data types and pointer types. There are no automatic type conversions, but there is a cast operator for each type.

Expressions

The expressions are quite C-like, permitting assignment expressions and the same arithmetic and comparison operators as in C. C’s bit operations are absent.

Control Constructs

Recursive functions, while, if-then-else, return, and exit constructs are present as in C. GPPL lacks for, goto, and labels.

5.2.2 Implementation: gc

gc is a compiler for GPPL implemented using the Ox system.

Target Language

For portability of gc-generated code, gc writes programs in a very small subset of C. That subset corresponds very closely to typical assembly languages. In each gc-generated program, there are declared only two variables (a stack pointer and a frame pointer), and only one function (main).

gc’s Source Code

The following table indicates gc’s organization.
### General Design

Synthesized attribute instances of terminal nodes hold the low-level information upon which gc's semantic analysis is based. About half of the semantic checks are done pursuant to attribute evaluation, the rest being interleaved with code generation during a single postdecoration postorder traversal.

#### 5.2.3 Performance of gc

To test the efficiency and correctness of gc and the C code it generates, there were written GPPL programs for the following:

- integer arithmetic functions: prime number generation, prime factorization, factorial, fibonacci.
- floating point functions such as: square root, logarithm, exponentiation.
- sorting: selection sort, heapsort, quicksort.
- solutions to some mathematical puzzles.

A large GPPL file was needed to test the compilation speed of gc, so five copies of each of the test files mentioned above were gathered into a single file of about 64 kilobytes. As it is a composite of many test files, this large file contains a representative mixture of GPPL's syntactic constructs.
Space Cost of Parse-tree Construction

By generating gc using Ox’s -u command-line option, gc was made to produce the following statistics when compiling the above-mentioned large test file.

| Size of input: | 17062 tokens |
| Parse-tree Storage Maximum Usages: | used/allocated |
| n: bytes of node storage: | 1363592/1400000 |
| c: number of child pointers: | 49374/50000 |
| r: number of dependee counts: | 163138/170000 |
| Maximum Parse-tree Storage Used: | 1724226 bytes |

- 20710 attribute instances in 17062 leaf nodes
- 142428 attribute instances in 32313 interior nodes
- 163138 attribute instances in 49375 parse-tree nodes
- 121450 attribute instances solved during decorations
- 121450/163138 = 0.744
- 5262 attributeless leaf nodes

Section 6.1 discusses possible improvements of the space efficiency of Ox-generated evaluators.

Relative Time Costs of Compilation Phases

Measuring Technique Ox has features that facilitate construction of programs that forego selected phases of the usual compilation sequence:

- By using Ox’s -S command-line option (section 1.12.3), it is easy to produce, from the source code for gc, an ordinary Lex/Yacc program that performs lexical and syntax analysis but does not build a parse tree.
- The technique of passing the Y-file and L-file through Ox twice, first using -S, then without using -S, can be used to generate a program that builds a dummy parse tree (one lacking attribute instances). The
following command sequence produces files \texttt{lex.yy.o} and \texttt{y.tab.o} that can be linked with \texttt{gc}'s other object files to produce such a parse-tree-building program.

\begin{verbatim}
  ox -S gppl.Y gppl.L
  mv ox.out.y tempYfile
  mv ox.out.l tempLfile
  ox tempYfile tempLfile
  yacc -d ox.out.y
  lex ox.out.l
  cc -c lex.yy.c
  cc -c y.tab.c
\end{verbatim}

- The \texttt{-n} command-line option (section 1.12.4) and the \texttt{@disable} construct (section 1.9) can be used to generate a program that builds a parse tree and performs a topological sort but does no semantic analysis or code generation.

- Inserting into the Y-file a declaration of the form:
  \begin{verbatim}
  @traversal identifier
  \end{verbatim}
  causes a dummy postdecoration traversal (i.e., one for which there are no traversal actions), and thus facilitates measurement of the overhead cost of traversal, as distinct from the cost of execution of the traversal actions themselves.

**Results** The above techniques were used to produce from the \texttt{gc} source code six programs whose running times indicate the costs of the various compilation phases. The table below shows for each of the six programs the set of functions performed by that program and the running time of that program relative to the running time of program 1. To obtain the average running times, the programs were executed at least 60 times each in random order.
<table>
<thead>
<tr>
<th>function</th>
<th>program number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1  2  3  4  5  6</td>
</tr>
<tr>
<td>program performs given function?</td>
<td></td>
</tr>
<tr>
<td>scanning &amp; parsing</td>
<td>● ● ● ● ● ●</td>
</tr>
<tr>
<td>parse-tree construction</td>
<td>● ● ● ● ● ●</td>
</tr>
<tr>
<td>topological sort</td>
<td>● ● ● ● ● ●</td>
</tr>
<tr>
<td>attribute evaluation</td>
<td>● ● ● ● ● ●</td>
</tr>
<tr>
<td>dummy traversal</td>
<td>● ● ● ● ● ●</td>
</tr>
<tr>
<td>code-generation traversal</td>
<td>● ● ● ● ● ●</td>
</tr>
<tr>
<td>average running time</td>
<td>average running time</td>
</tr>
<tr>
<td>(CPU seconds)</td>
<td>1.3  3.1  5.5  5.7  6.3  8.0</td>
</tr>
<tr>
<td>(CPU time relative to program 1)</td>
<td>1.0  2.4  4.2  4.4  4.8  6.1</td>
</tr>
</tbody>
</table>

From the above table, it can be seen, for instance, that parse-tree construction alone took 1.4 (i.e., $2.4 - 1.0$) times as long as scanning and parsing, and that the topological sort alone took 1.8 (i.e., $4.2 - 2.4$) times as long as scanning and parsing.

### 5.2.4 Remark

As GPPL and ge evolved, it was remarkably easy to introduce or abandon attributes and grammar symbols: the automatic checks for consistency and completeness (section 1.5.1) and automatic generation of code for parse-tree management afforded a singular confidence in modifying the design and implementation.
Chapter 6

Future Work

Presently Ox represents the core of a Yacc/Lex/C-based language for expressing attribute grammars, and shows that such a language can be implemented efficiently. This chapter suggests extensions to the language and improvements in the implementation. In estimating the benefits of proposed improvements, reference is made to the gc example of section 5.2.3.

6.1 Easy Improvements in Storage Efficiency

This section describes some modifications that would bring large gains in space efficiency, yet require no major changes to Ox’s semantic analysis functions. These changes would require major changes to the grammar-independent part of the generated evaluator, but would have little effect on its size or complexity.

6.1.1 Dependee Counts as Unsigned Nybbles

The evaluators generated under the present implementation keep a dependee count for each attribute instance. This constitutes about 10% of the parse-tree-related storage in the above-mentioned gc example.

Presently, dependee counts are all of type unsigned char unless there is a count that is to be initialized to something greater than 255, in which case they are all of type unsigned short. Ox could be made such that if no count is to be initialized to more than 15, each count would be stored

71
in half of an unsigned char. This would afford a 5% overall saving in the gc example. Experiments indicate that the additional bit operations would entail an overall slowdown of at most a few percent.

6.1.2 Eliminating Attribute-less Leaf Nodes
The parse tree for the gc example had 5262 leaves that lacked attribute instances. Modification of Ox such that no storage is allocated for those leaves would save twenty bytes (see below) for each one, about 6% of parse-tree storage.

6.1.3 Removing Indirection from Parse-Tree Nodes
The present implementation produces evaluators that build parse trees whose nodes have components for:

- a pointer to the node’s parent
- a pointer into the space of child pointers (marks the beginning of the list of the node’s children)
- a pointer into the space of dependee counts (marks the beginning of the list of the node’s dependee counts)
- the serial number of the rule applied at the node
- the position (on the RHS of the parent rule) corresponding to the node

Instantiations of this data structure are referred to as generic nodes. The data types of the components are determined statically. The rule number can be stored as a char or a short (typically one or two bytes), and the position is normally stored as a char. Pointers typically occupy four bytes each, so each generic node occupies at least fourteen bytes. Due to alignment requirements, generic nodes occupy sixteen bytes each for typical architectures. In addition, in the child-pointer space, there is a pointer to each node except the root. Thus each non-root node accounts for twenty bytes, not including its dependee counts and attribute instances. Storage for attribute instances is allocated from the same space as is storage for generic nodes, storage for each
generic node being followed by storage for its attribute instances, resulting in an alternating pattern.

As described in section 3.11.2, the practice of storing dependee counts contiguously in a separate space facilitates cycle detection. Storing lists of child pointers in a separate space is a vestige of an earlier design. The child-pointer pointers and the dependee-count pointers of the generic nodes could be eliminated by storing child pointers and dependee counts in the same space as the generic nodes and attribute instances, for a saving of up to eight bytes per node.

6.1.4 Calculation of Space Savings

The statistics generated in the ge example together with assumptions about data sizes and alignment requirements permit an exact calculation of the savings promised by the three design modifications of the preceding sections when applied to that example.

In the example, 1,363,592 bytes were used for storing generic nodes and attribute instances. As indicated above, sixteen bytes were used for each of the 49,375 generic nodes. Thus \((1,363,592 - (49,375 \times 16))\) or 573,592 bytes were required for storing attribute instances. The design modifications do not affect this number.

The proposed modifications eliminate 5262 generic nodes, and each generic node is to require eight bytes, so \((49,375 - 5262) \times 8\) or 352,904 bytes are needed for generic nodes.

Reducing the number of nodes reduces the number of child pointers. \(((49,374 - 5262) \times 4)\) or 176,448 bytes are needed for child pointers.

Examination of ge’s source code [Bischoff 92] reveals that no symbol in ge’s grammar has more than eight attributes, and that no dependee count is initialized to more than 15. Therefore no node needs more than four bytes for storing its dependee counts. To ensure proper alignment, the dependee count storage area for each node is allocated in chunks of four bytes. Each node thus requires exactly four bytes for storing its dependee counts. Thus \(((49,375 - 5262) \times 4)\) or 176,452 bytes are needed for dependee counts.

All parse-tree storage needed under the modified design has been counted in the four paragraphs above. It follows that \((573,592 + 352,904 + 176,448 + 176,452)\) or 1,279,396 bytes are occupied by parse-tree storage.
This saves \((1,724,226 - 1,279,396) \div 1,724,226\) or about 26\% of the space required under the present implementation.

6.2 Grammar-wide Semantic Analysis

6.2.1 Static Circularity Test

In the present implementation, it is possible to build an evaluator for a circular grammar. It seems unlikely that such circularity would remain undiscovered after even moderate exercise of the generated evaluator, but a static circularity test might prevent some backtracking in the design process.

Experiments have shown that with the present scheme (section 3.11.2) for storing dependee counts, the time cost of the dynamic circularity test is almost negligible. With the space-saving design of section 6.1.3 the time cost of a dynamic circularity test would be greater, since a stack and a tree traversal would be required, although it would still be relatively small.

6.2.2 Solving More Instances at Reduction Time

As discussed on page 45, some attribute instances are solved as part of Yacc and Lex actions, and are not involved in the dynamic phase of the topological sort. About one-fourth of the instances in the gc example were so solved. This improves the time efficiency of evaluation. Considering the dependency relations of the grammar as a whole (as opposed to analysis of each rule in isolation) promises to increase the proportion of instances solved without involvement in the dynamic topological sort. Ideally, such instances need no dependee counts, so there may also be space savings in this aspect of optimization.

6.2.3 Global Static Topological Sorting

The analyses mentioned in the two preceding sections (a static circularity test and finding instances solvable in Yacc or Lex actions) might be implemented so as to utilize the same data structures and to be interleaved in execution.

The most important feature is the maintenance of dependee counts not only for output attribute occurrences, but also for attributes. The dependee
counts for an output occurrence may be called a *local dependee count*. As in the present implementation (see section 3.2), each local dependee count is initialized with the number of attribute occurrences upon which its occurrence depends. A *global dependee count* is the dependee count for a given attribute, and initially holds the grammar-wide number of output occurrences of that attribute.

As in the present implementation, a zero value for a local dependee count would mean that it is possible to solve all instances corresponding to its occurrence. After a local dependee count becomes zero, the global dependee count of that occurrence’s attribute is decremented. Decrementing a global dependee count to zero means that all instances of all occurrences of that attribute are solvable.

Further analysis is expected to reveal that this technique is not capable of deciding general noncircularity. Hopefully it gives rise to a test for some “natural” kind of noncircularity.

### 6.3 Extensions to Ox Syntax and Semantics

#### 6.3.1 Dependees for Synthesized Occurrences of Tokens

There are no great implementation problems involved in permitting synthesized occurrences of tokens to depend upon other synthesized occurrences. It is important that such synthesized occurrences be solved at lookahead time, since they are usually defined in terms of the contents of `yytext`. Thus the dependee closures (page 62) of synthesized occurrences of tokens must not contain inherited occurrences.

In Ox’s grammar, the productions for handling attribute definitions in the L-file(s) are distinct from those that handle attribute definitions in the Y-file. Using the same productions for both cases would add regularity to the language and simplify the implementation. A context-sensitive flag would tell Ox’s semantic analysis phase whether it is processing part of the Y-file or part of an L-file.
6.3.2 L-file(s) not Based on Lex

It might sometimes be convenient for Ox to take as an L-file a scanner consisting of hand-written C code (rather than Lex regular expressions) and attribute definition sections.

This would be implemented by endowing Ox with an alternate scanner (see page 60) that simply looks for attribute definition sections following return statements. The present scanner for L-file s takes no special notice of strings having return, {a, and }@ as substrings when they occur in the context of Lex regular expressions.

6.3.3 Defaults for Attribute Definitions

Often a Y-file has many attribute definitions that function only to copy an instance belonging to one node to a like-named instance belonging to the node’s parent or child. This situation is conspicuous when contextual information is moved leafward via inherited attributes. Ox could be built such that if a RHS inherited occurrence is undefined, and the LHS has a like-named occurrence, the undefined occurrence is by default a copy of the like-named LHS occurrence.

The present version of Ox checks that no occurrence lacks a definition or has more than one definition. This policy detects the most common errors, but it can be argued that the ability to specify default definitions according to certain patterns reduces tedium and improves safety. Such features have been implemented in various attribute-grammar compilers [Lorho 88], and their implementation in future versions of Ox presents no difficulty.

6.3.4 Automatic Construction of Syntax Trees

Often an attribute grammar has grammar symbols whose importance is mainly syntactic, rather than semantic. Semantic information is merely “copied through” nodes labeled by those symbols, as indicated in the previous section. Significant saving of space could be made by “collapsing” branches of the tree so that such nodes are eliminated.

Provision for specification of syntax trees and definitions of their attributes presents an interesting language design and implementation problem in the context of Ox.
6.3.5 Traversals at Reduction Time

Presently, traversals are performed only after the final decoration. In many cases, a postdecoration traversal is the sole impediment to pruning. Endowing Ox with constructs for specifying traversals of stacked subtrees could obviate postdecoration traversal of those subtrees and might permit significant space savings via pruning. The implementation of such constructs does not appear difficult, but special care should be taken in designing their syntax and semantics.

6.3.6 Repetition of Traversals

One approach to translating a program is to build and decorate a parse tree, then execute the program by traversing the decorated tree. For this semi-interpretive approach, execution of iterative and recursive control structures involves repeated traversals of subtrees.

By instituting a new kind of dynamic traversal modifier (recall section 1.9.2), Ox could be made to support the above-mentioned approach. A given traversal would be performed once, repeatedly, or not at all, depending upon the value of an associated C/Ox expression. Since the attribute instances of a node and its children are accessible during traversals, instances could be used for loop counters and predicates. The proposed Ox constructs could be made to support for, while, and do-while semantics.

Function-call semantics could be supported by permitting attributes of the special type “pointer-to-subtree”, and permitting “random-access” traversals of the indicated subtrees.

6.4 Facility for Interactive Debugging

Ox could be made to generate evaluators each capable of using X windows to display the stack of subtrees. Clicking on a node would call a pop-up display to show the values of the node’s attribute instances. The system would be a “grammar-level” debugger, with features analogous to those of source-level debuggers such as dbx. Features would include the ability to view the stack of subtrees:

- after the next $n$ stack operations, for any $n$
• after each stack operation that pushes any of a specified set of grammar symbols

• after each stack operation involving any of a specified set of rules

6.5 Support for Other Languages

The present version is made to work with the most widely-available versions of Yacc, Lex, and C (i.e., those ordinarily received bundled with Unix systems). Small differences may inhibit its use with similar languages such as Bison, Flex and C++. A single version compatible with all of these can be built easily, given experience and small changes to Ox.

6.6 Multiprocessing

Disjointness of storage spaces for disjoint subtrees suggests parallel processing: one processor for each subtree, each with its own memory region.
Acknowledgement

This is to thank my adviser Kelvin D. Nilsen for suggesting the use of Knuths’s algorithm for topological sorting using dependee counts, for suggesting improvements regarding some important implementation details, and for critical readings and suggestions for revisions of this paper.

Thanks also to Gary T. Leavens and Don L. Pigozzi for reviewing this paper and serving on my graduate committee.
Bibliography


[Lorho 88] Pierre Deransart, Martin Jourdan, and Bernhard Lorho, *Attribute Grammars: Definitions, Systems, and Bibli-


