Ox:

An Attribute-Grammar Compiling System
based on Yacc, Lex, and C:

User Reference Manual

by Kurt M. Bischoff

November 14, 1993

Contents

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 can specify a language using the familiar languages of Lex and Yacc, or take an existing Lex/Yacc specification, and add semantics to the language by augmenting the specification files with declarations and definitions of typed attributes of parse-tree nodes.

That specification constitutes an *attribute grammar*, and from it Ox can automatically generate 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 can 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

[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.

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 [2] 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 can 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.

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. [3] 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 lexical analyzer [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 specifi-

[2] The general descriptions in this manual assume the use of a Lex-based lexical analyzer. It is possible, however, to use Ox with lexical analyzers hand-written in C: details are given separately (in appendix A).

[3] "Yacc", "Lex", and "C" can, in this manual, be taken to mean "Yacc or Bison", "Lex or Flex", and "C or C++", respectively.

cations, the complete texts of which appear in sections 14, 15, and 16.

Within Ox-specific constructs, C-style and 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`.

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:

```
@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:

```
* : ; , char short int long float double
    signed unsigned struct union enum
```

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.

All uses of the Yacc reserved words `%token`, `%left`, `%right`, and `%nonassoc` must precede all of the attribute declarations.

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.

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*). The *right-hand side* (*RHS*) comprises 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 3 and the rule:

```
num     :       bitlist DOT     bitlist
```

the attribute occurrence `scale` of the leftmost appearance of `bitlist` is denoted in Ox code as `bitlist.0.scale`, while the attribute occurrence `scale` of the rightmost appearance of `bitlist` is denoted `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.

5 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.

5.1 Inherited vs. synthesized attributes

An attribute occurrence o in 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 in 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 nonterminals, 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 (section 3.1), 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.

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 `@{` 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. [4] Conceptually, an attribute definition has a *dependency part* and an *evaluation part*, but syntactically,

[4] 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.

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 `@}`.

5.2.1 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:

```
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 makes explicit the 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 called *dependees* 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 arithmetic, 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.

5.2.2 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.

```
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 annunciator `@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.

5.2.3 Mixed mode

Mixed mode attribute definitions are announced by the reserved word `@m`. 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.

```
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;
        @}
        ;
```

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 `return`ed token.

Note that each point of `return` of `yylex` must be *explicit* in the sense that the text must bear the C reserved word `return`. In particular, `return`s must not be done within C macros, unless the L-file is passed through the C preprocessor prior to processing by Ox. Guaranteeing this property of `yylex` is the responsibility of the Ox user—it is not checked by Ox.

5.3.1 Generality of Ox

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 (wherein `CONST`'s only synthesized attribute is `val`):

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

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

5.3.2 Ox adaptation to Lex's line-oriented syntax

When Ox is processing the L-file and has recognized a rule (i.e., the object of a `return` statement), if the `return`ed token has synthesized attributes, Ox looks for an attribute reference section following the `return` statement. Ox's rules for recognizing attribute reference sections in the L-file are adapted from the way Lex actions are terminated: Ox gives up looking for an attribute reference section when it pairs the rightmost right curly brace in the action with the leftmost left curly brace, or when it encounters a newline unprotected by curly braces. Newlines are insignificant inside attribute reference sections.

Examples of correct and incorrect syntax are shown below. All of the correct forms shown are semantically equivalent to one another.

- incorrect (attribute reference section appears to the right of the rightmost curly brace):

```
[a-zA-Z]+ { count(); return ID; } @{ @ID.name@ = id(); @}
```

- incorrect (attribute reference section not part of rule, since the Lex action is terminated by an unprotected newline):

```
[a-zA-Z]+ count(); return ID;
          @{ @ID.name@ = id(); @}
```

- correct:

```
[a-zA-Z]+ { count(); return ID;  @{ @ID.name@ = id(); @} }
```

- correct:

```
[a-zA-Z]+ return ID; @{ count(); @ID.name@ = id(); @}
```

- correct:

```
[a-zA-Z]+ { count(); return ID;
            @{ @ID.name@ = id(); @}
          }
```

- correct:

```
[a-zA-Z]+ count(); return ID;  @{
                                 @ID.name@ = id();
                               @}
```

5.3.3 Resolution of ambiguity regarding token returned

A slight difficulty arises in rules like

```
return(yytext[0]);
```

and

```
return(cond ? TOKEN1 : TOKEN2);
```

for which Ox cannot determine at evaluator-generation time which token will be `return`ed.

In the first case, wherein no declared token or character constant is recognized in the `return`ed expression, Ox assumes that the token `return`ed has no attributes, and issues a warning like:

```
ox: scan.l: warning: line 8: ambiguous form of return of token.
  unknown node type--assuming 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 *leftmost* in the expression. Ox issues a warning like:

```
ox: scan.l: warning: line 8: ambiguous form of return of token.
  multiple tokens in object of return statement.
```

The above warnings should be taken seriously, because the conditions of which they warn can result in the generated evaluator attempting to access attribute instances that are nonexistent or of the wrong type. These kinds of warnings are most often seen when first converting an existing Yacc/Lex translator to Ox.

A condition causing one of the above-described warnings may be tolerated if the Ox user verifies that for the rule (i.e., object of the `return` statement) in question:

- all of the tokens that can be `return`ed for the rule are contained in the grammar-symbol list (section 3) of a single attribute declaration.
- no token that can be `return`ed for the rule appears in a grammar-symbol list of an attribute declaration.

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.

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.

7 Temporal behavior of Ox-generated evaluators

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.

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.

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.

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 lexical analyzer. 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 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 (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 9.

9 Postdecoration traversals

The idea of decoration was described in section 7.3. *Postdecoration* refers to any time after the *final* decoration of the parse tree, which follows parsing of a correct input. This section shows how the Ox user can cause *postdecoration traversals*, each of which permits access (in a user-specified order) to the tree's attribute instances.

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 "oxout.h"
%}

%%
[ \n\t\f]*     ;
[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 "oxout.h"
#include <stdio.h>
%}

%%
expr    :       expr    '*'     expr                            /* rule 1 */
                @{ @LRpre  printf(" * "); @}

        |       expr    '+'     expr                            /* rule 2 */
                @{ @LRpre  printf(" + "); @}

        |       '('     expr    ')'                             /* rule 3 */

        |       CONST                                           /* rule 4 */
                @{ @LRpre  printf(" %d ",@CONST.val@); @}
        ;
%%


main()
  {yyparse();
   printf("\n");
  }
```

The sequence: `@traversal @lefttoright @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 subtree rooted at a child of the node is traversed, the leftmost subtree first. 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. No action is performed during a traversal of a subtree that consists of a terminal node.

9.2 General description

9.2.1 Traversal specifications

The Ox programmer may place in the declarations section (the part before the first `%%` mark) of the Y-file one or more *traversal 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 may contain the following *traversal specifiers* (in any order):

- at most one of: `@postorder`, `@preorder`
- at most one of: `@lefttoright`, `@righttoleft`
- 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 `@righttoleft` 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`.

9.2.2 Traversal action specifications

In addition to attribute definitions (section 5.2), the attribute reference sections of the Y-file may contain *traversal 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 `LRpre` and `LRpost` as above. Then in the code fragment:

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

the attribute reference section has four traversal action specifications and no attribute definitions. Each specification is announced by either `@LRpre` or `@LRpost`. Each of the `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 `@revorder (1)` as a dynamic traversal modifier. A dynamic traversal modifier is either `@revorder` or `@revdirection`, followed by a parenthesized expression that conforms to C syntax, except that it may refer to the rule's attribute occurrences. `@revorder` and `@revdirection` may each occur at most once in a given traversal action specification. If `@revdirection` 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 `@revorder` *expr* means roughly "reverse order `if` *expr*". When the `LRpre` traversal reaches a node at which the rule `s :  expr` is applied, the expression `(1)` is evaluated, and because it is nonzero, the third traversal action, which prints a line feed, is executed as if `LRpre` were a postorder traversal, i.e., *after* the recursive traversal of the subtree rooted at the node's sole child. The execution of the fourth traversal action, `printf("prefix:  ");` is not affected by any dynamic traversal modifier, and occurs according to `LRpre`'s (static) specification, i.e. *before* the traversal of the child subtree.

When the `LRpost` traversal reaches a node at which `s :  expr` is applied, the second traversal action is executed, the traversal proceeds to the child subtree, then the first traversal action is executed.

The preceding description is generally sufficient for understanding post-decoration traversals, but appendix B contains a pseudocode description that describes the behavior somewhat more formally.

Facility for inorder traversal is to be implemented in future versions of Ox.

10 Ox macros

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

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.

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.

10.3 Example

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

```
   ⋮

@macro exprdefs(op,)
  @i @expr.1.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

   ⋮

%%

   ⋮

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.

11 Automatic generation of copy rules

Often a Y-file has 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. Large attribute grammars tend to have many such definitions, which are sometimes called *copy rules*. The situation is conspicuous when contextual information is moved leafward via inherited attributes.

The Ox user may place in the declarations section of the Y-file the construct:

`@autoinh` <ID_list>

where <ID_list> is a whitespace-separated list of attribute names. Suppose that `attrbID` is such an attribute name, and the above construct is followed by an `@attributes` declaration whereby `attrbID` is declared as an attribute of the grammar symbol `gSym`. Then Ox knows that `attrbID` is an inherited attribute of `gSym`. Further, for any rule having `gSym` on the RHS, Ox searches that rule's attribute reference section for definitions of the RHS occurrences of `attrbID`. When such a definition is missing, Ox checks whether the LHS has an occurrence of `attrbID`. If so, Ox generates definitions that copy that LHS occurrence to each RHS `attrbID` occurrence that lacks a definition. If there is no such LHS occurrence, Ox issues an error message.

There is an analagous construct for automatic generation of definitions of synthesized occurrences:

`@autosyn` <ID_list>

When the `@autosyn` construct is used, Ox tries to supply missing definitions of synthesized occurrences by searching the RHS for same-named occurrences. If exactly one such RHS occurrence is found, Ox generates a definition to copy it to the LHS, otherwise there is an error.

The above-described constructs have a global character in that a single `@autosyn` or `@autoinh` declaration can easily be used to supply missing definitions for all occurrences of attributes of a given name. These reserved words may be used in a more conservative way that generates missing definitions only for occurrences belonging to a selected set of grammar symbols:

Attribute declarations are written as usual, except that `@autoinh` or `@autosyn` may appear before the attribute's type specifier (i.e., after { or `;`). Where <ID_list> is the usual comma- separated list of attribute names, and `attrbID` is a member of <ID_list>:

```
@attributes    {
                  ⋮
                  @autoinh <typespec> <ID_list> ;
                  ⋮
               }
               <grammar_symbol_list>
```

declares `attrbID` as an inherited attribute whenever it occurs in a symbol in <grammar_symbol_list>. Further, this instructs Ox to attempt to supply missing definitions of such occurrences by copying from the LHS.

`@autosyn` may be used locally in an analagous manner.

For safety in the use of `@autosyn` and `@autoinh`, there is provided the `@warn` reserved word. When it immediately follows `@autosyn` or `@autoinh`, Ox issues a warning for each definition supplied by virtue of the preceding `@autosyn` or `@autoinh`. `@warn` is mainly to be used when modifying the attribute grammar.

11.1 Example

The following code fragment in the declarations section of the Y-file:

```
   ⋮

@autoinh env

@attributes {struct env *env;
             regNumType maxRegNum;
            }
            execElem statement

@autosyn maxRegNum

@attributes {struct env *env;
             @autoinh regNumType regNum;
             regNumType maxRegNum;
             struct sym *formParamList;
             struct sym *func;
             lineNumType line;
            }
            actParamList

@attributes {struct env *env;
             @autosyn @warn struct sym *varLoc,*funcLoc;
             regNumType regNum;
             regNumType maxRegNum;
            }
            block blockElemList

   ⋮
```

causes Ox to attempt to automatically supply missing definitions for occurrences of:

- `env` for `execElem`, `statement`, `actParamList`, `block`, and `blockElemList`
- `maxRegNum` for `actParamList`, `block`, and `blockElemList`
- `regNum` for `actParamList`
- `varLoc` for `block` and `blockElemList`, with warning
- `funcLoc` for `block` and `blockElemList`, with warning

12 File-level organization of Ox evaluators

12.1 Conventions of naming Ox output files

Ox translates the Y-file into a file destined for processing by Yacc, given the name `oxout.y`. The L-files are translated into files destined for Lex. If there is exactly one L-file, its corresponding output file is named `oxout.l`. If there is more than one L-file, the corresponding outputs are named `oxout0.l`, `oxout1.l`, `oxout2.l`, etc.

12.2 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 `y.tab.c` and `lex.yy.c`. The two files may then be compiled separately if the line

```
#include "y.tab.h"
```

is placed in a C-code sections of the Lex input specification. The two resulting object files can then be linked to produce an executable file.

12.3 Combined use of Ox, Yacc, and Lex

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

```
#include "oxout.h"
```

is placed in the Y-file.

12.4 Typical command sequences

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

```
ox ev.Y ev.L
yacc -d oxout.y
lex oxout.l
cc -c y.tab.c
cc -c lex.yy.c
cc -o ev y.tab.o lex.yy.o -ll -ly
```

The following command sequence does a one-step compilation.

```
ox -h ev.Y ev.L
yacc oxout.y
lex oxout.l
cc y.tab.c -ll -ly
```

13 Command-line options and other points

This section describes some fine points, mostly related to Ox command-line options. Use of those options is summarized in appendix D.

13.1 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 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 `yyerror` upon any syntax error, and the designer should write `yyerror` such that `yyyerror` is executed at least once each time `yyerror` 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`.

13.2 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 `-a`N 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.

13.3 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 `oxout.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 `oxout.*` contain neither Ox constructs nor the usual Ox-generated parse-tree management code.

13.4 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.

13.5 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 `-Y`aN 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 (see appendix D).

The evaluator designer can easily build an evaluator that accepts the same `-Y`aN command-line options accepted by Ox: By specifying the `-YY` option on Ox's command line, Ox is instructed to declare in the generated evaluator a function `yyyCheckForResizes` that can read `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 `main` program according to the following form:

```
   ⋮

void yyyCheckForResizes();

   ⋮

main(argc,argv)
  int argc;
  char *argv[];
  {

   ⋮

   /* This is executed before calling yyparse */
   yyyCheckForResizes(argc,argv);

   ⋮

   yyparse();

   ⋮

  } /* main */
```

13.6 Parse tree statistics

Placing `-u` on 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.

13.7 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 `-X`aN to make N the size of data structure a.

14 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 `'+'`:

```
%{
/* expr.L:  L-file for a simple expression language */
#include "y.tab.h"
#include "oxout.h"
%}

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

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 "oxout.h"
long sVal;
%}

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

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 oxout.y
lex oxout.l
cc -c y.tab.c
cc -c lex.yy.c
cc -o calc y.tab.o lex.yy.o -ly -ll
```

15 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:

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

%%
[0]             return ZERO;
[1]             return ONE;
\.              return DOT;
[\n\t\v ]       ;
.               {fprintf(stderr,"illegal character\n");
                 exit(-1);
                }
```

Here is the text of the Y-file:

```
%token ZERO ONE DOT

@attributes {float value; int scale;}               bit
@attributes {float value; int scale,length;}        bitlist
@attributes {float value;}                          num

%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 @bit.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 @bit.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("%30.15f\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));
  }
```

Construction of the above evaluator follows the separate compilation approach described in section 12.

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.

16 Example: translation 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:

```
%{
/* L-file for translation of infix expressions */
#include "y.tab.h"
#include "oxout.h"

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

%}

%%
[ \n\t\f]*              ;
[0-9]+\.?[0-9]*         return(CONST); @{ @CONST.lexeme@ = lexeme(); @}
[A-Za-z_][A-Za-z_0-9]*  return(ID);    @{ @ID.lexeme@ = lexeme();    @}
\(                      return('(');
\)                      return(')');
\+                      return('+');
\*                      return('*');
\/                      return('/');
\-                      return('-');
%%
```

The first traversal performed is named `LRpre`, and the second is named `LRpost`. By default, both are left-to-right traversals. `LRpost` is a postorder traversal by default. `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 "oxout.h"
#include <stdio.h>
%}
```

```
%%
s       :       expr
                @{ @LRpost printf("\n");
                   @LRpost @revorder (1) printf("postfix:  ");
                   @LRpre  @revorder (1) printf("\n");
                   @LRpre  printf("prefix:   ");
                @}
expr    :       expr    '*'     expr
                @{ @LRpost printf(" * ");
                   @LRpre  printf(" * ");
                @}
        |       expr    '+'     expr
                @{ @LRpre  printf(" + ");
                   @LRpost printf(" + ");
                @}
        |       expr    '/'     expr
                @{ @LRpost printf(" / ");
                   @LRpre  printf(" / ");
                @}
        |       expr    '-'     expr
                @{ @LRpost printf(" - ");
                   @LRpre  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();
  }
```

A Using Ox with non-Lex lexical analyzers

A.1 Default context-sensitivity of L-file preprocessing

Unless instructed otherwise, Ox searches each L-file for `return` statements in the context of C-coded Lex actions. Since the string `return` must be ignored outside of that context (for instance, in a Lex regular expression), the default behavior of Ox-preprocessing is to assume that the L-file conforms to the syntax of a (possibly Ox-augmented) Lex file. Thus Ox recognizes the three `return` statements in the following (unaugmented) fragment of an L-file:

```
  ⋮
renames   return(TK_RENAMES);
return    return(TK_RETURN);
reverse   return(TK_REVERSE);
  ⋮
```

as points of `return` of tokens by `yylex`. Ox's sensitivity to the context of the string `return` in the second Lex regular expression above prevents its erroneous recognition as a point of `return` from the lexical analyzer.

A.2 Ox-preprocessing of C-coded lexical analyzers

Ox always ignores the string `return` in the context of C/C++ comments and string constants. When given the `-G` command-line option preceding the name of an L-file, it ignores `return` *only* in those contexts. Thus a file containing C/C++ code may be augmented with attribute reference sections and input to Ox as an L-file. The occurrences of the string `return` must coincide exactly with `return`s of tokens.

A.2.1 Example

Suppose it is desired to convert to Ox a translator that uses the following C code for its lexical analyzer:

```
#include <stdio.h>
#include <string.h>
#include "y.tab.h"

#define bufsize 80
char buf[bufsize];
char *lexBuf;

char *lexeme(inString)
  char inString[];
  {return strcpy((char *)malloc(1+strlen(inString)),inString);}

int yylex()
  {char *bufp = buf;

   while ((*bufp = getchar()) != EOF)
     {if (bufp == (buf + bufsize - 1))
         {fprintf(stderr,"exceeded buffer\n"); exit(-1);}
      if ((*bufp == ' ')  || (*bufp == '\n') ||
          (*bufp == '\t') || (*bufp == '\f')
         )
         {if (bufp == buf) continue; else break;}
      if (!isalnum(*bufp)) {fprintf(stderr,"illegal character\n"); exit(-1);}
      bufp++;
     }
   if (bufp != buf)
      {*++bufp = '\0';
       lexBuf = lexeme(buf);
       bufp = lexBuf;
       if (isalpha(lexBuf[0]))
          {while (*bufp != '\0')
              if (isdigit(*bufp++))
                 {fprintf(stderr,"illegal string\n"); exit(-1);}
           return (IDENT);
          }
       if (isdigit(lexBuf[0]))
          {while (*bufp != '\0')
              if (isalpha(*bufp++))
                 {fprintf(stderr,"illegal string\n"); exit(-1);}
           return (ICONST);
          }
      }
   return 0;
  }
```

The C reserved word `return` occurs in the file exactly four times. Only two of these occurrences correspond to `return`s of tokens by the lexical analyzer. The approach is to excise from the file a section of code containing those two occurrences (and no others), place that code section it in a separate file, and submit the new file to Ox as a non-Lex L-file, by using the `-G` option. Ox translates the new file and places it on `oxout.l`. The excised code is replaced in the original file by the line:

```
#include "oxout.l"
```

Here is the L-file, which has been augmented by two attribute reference sections:

```
if (isalpha(lexBuf[0]))
   {while (*bufp != '\0')
      if (isdigit(*bufp++))
         {fprintf(stderr,"illegal string\n"); exit(-1);}
    return (IDENT);  @{ @IDENT.string@ = lexBuf; @}
   }
if (isdigit(lexBuf[0]))
   {while (*bufp != '\0')
      if (isalpha(*bufp++))
         {fprintf(stderr,"illegal string\n"); exit(-1);}
    return (ICONST); @{ @ICONST.string@ = lexBuf; @}
   }
```

Here is the skeleton of the lexical analyzer, which now `#include`s the files `oxout.h` and `oxout.l`:

```
#include <stdio.h>
#include <string.h>
#include "y.tab.h"
#include "oxout.h"

#define bufsize 80
char buf[bufsize];
char *lexBuf;

char *lexeme(inString)
  char inString[];
  {return strcpy((char *)malloc(1+strlen(inString)),inString);}

int yylex()
  {char *bufp = buf;

   while ((*bufp = getchar()) != EOF)
     {if (bufp == (buf + bufsize - 1))
         {fprintf(stderr,"exceeded buffer\n"); exit(-1);}
      if ((*bufp == ' ')  || (*bufp == '\n') ||
          (*bufp == '\t') || (*bufp == '\f')
         )
         {if (bufp == buf) continue; else break;}
      if (!isalnum(*bufp)) {fprintf(stderr,"illegal character\n"); exit(-1);}
      bufp++;
     }
   if (bufp != buf)
      {*++bufp = '\0';
       lexBuf = lexeme(buf);
       bufp = lexBuf;

#include "oxout.l"

      }
   return 0;
  }
```

B Traversal semantics

The behavior of postdecoration traversals was illustrated in the examples of section 9.2. In view of those examples, the C-like pseudocode in this appendix holds no surprises, but describes such behavior somewhat more formally. The traversals are carried out by a single call of `doTraversals` (below) after the final decoration.

```
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  */
   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 asociated with @revdirection) == 0)
               direction = staticDirection(T);
           else if (staticDirection(T) == LEFTTORIGHT)
               direction = RIGHTTOLEFT;
           else
               direction = LEFTTORIGHT;
       }
   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 == LEFTTORIGHT)
       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;
  }
```

```
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);
  }
```

C List of reserved words and reserved file names

The Ox reserved words are as follows:

```
@{
@}
@attributes
@autoinh
@autosyn
@disable
@e
@i
@lefttoright
@m
@postorder
@preorder
@revdirection
@revorder
@righttoleft
@warn
```

The following file names in the current directory are reserved for use by Ox:

`oxout.h`	(exactly when using `-h` option)
`oxout.y`	
`oxout.l`	(exactly when using exactly one Y-file)
`oxout0.l`, `oxout1.l`, `oxout2.l`, ...	(when using more than one L-file)

D Summary of command-line options

The Ox command line takes the form:

`ox` { *option* } *Y-file* [`-G`] *L-file* { [`-G`] *L-file* }

The `-G` annunciator and the *option*s are described as follows:

`-G` The filename which follows is that of a generic (i.e. non-Lex) L-file. This option is used when the Ox user prefers a scanner hand-written in C or C++. Except for attribute reference sections, the L-file must conform to C/C++ syntax. The `return` reserved word is recognized in any context other than comments and string literals. See appendix A.

`-C` Print the Ox copyright statement and disclaimer.

`-I` Generate code for C++ or ANSI/ISO C compilation.

`-S` Strip Ox-specific constructs from the Y-file and L-file(s) and place the pure Yacc and pure Lex results on `oxout*.*`. See section 13.3.

`-U` Show the form of the Ox command line.

`-V` Print the version number.

`-h` Produce an Ox header file `oxout.h` to be `#include`d in a code section (between `%{` and `%}`) in the Y-file or L-file(s). This permits one-step compilation of the parser and scanner(s). When this option is not used, Ox places the header information in each output file rather than in a separate header file. See section 12.3.

`-n` Generate an evaluator that determines an evaluation order and checks for cycles, but does not execute the code that evaluates attribute instances. See section 13.4.

`-u` Generate an evaluator that prints parse-tree memory-usage statistics for each input. See section 13.6.

`-a`N Set N as the generated evaluator's default structure-alignment size. By default, the default structure-alignment size is 4. See section 13.2.

`-Y`aN Where N is an integer and a is one of the alphabetic characters `n`, `c`, or `r`, change the default size of data structure a in the generated evaluator to size N. When such a data structure is exceeded, the generated evaluator issues an error message calling for use of a `-Y`aN option. The effects of the `-Y`aN options for the various values of a are as follows:

- `n` `-Yn`N causes allocation of N bytes for parse-tree nodes and attribute instances.
- `c` N is the maximum number of non-root nodes in the parse tree. `-Yc`N causes allocation of N `* sizeof(void *)` bytes for pointers to child nodes.
- `r` N is the maximum number of attribute instances in the parse tree. N bytes are allocated for this.

See sections 13.5 and 13.6.

`-YY` Enable the generated evaluator to understand the same `-Y`aN options as does Ox itself. See section 13.5.

`-X`aN Where N is an integer and a is an alphabetic character, change the default size of Ox's data structure a to size N. When such a data structure is exceeded, Ox issues an error message calling for use of a `-X`aN option.

References

[Johnson 75] Stephen C. Johnson, *Yacc: Yet Another Compiler Compiler*, Computing Science Technical Report No. 32, AT&T Bell Laboratories, Murray Hill, New Jersey, 1975. Reprinted as PS1:15 in *UNIX Programmer's Manual*, Usenix Association, 1986.

[Lesk 75] M.E. Lesk and E. Schmidt, *Lex–A Lexical Analyzer Generator*, Computing Science Technical Report No. 39, AT&T Bell Laboratories, Murray Hills, New Jersey, October 1975. Reprinted as PS1:16 in *UNIX Programmer's Manual*, Usenix Association, 1986.

[KR 88] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language, 2nd Ed.* Prentice-Hall, 1988.

[Waite 84] William M. Waite and Gerhard Goos, *Compiler Construction*, Springer-Verlag, 1984.

[Aho 86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[Jazayeri 75] M. Jazayeri, W.F. Ogden, and W.C. Rounds, *The Intrinsic Exponential Complexity of the Circularity Problem for Attribute Grammars*, Communications of the ACM, Vol. 18, No. 12, pp. 697-706, December 1975.

[Lorho 88] Pierre Deransart, Martin Jourdan, and Bernhard Lorho, *Attribute Grammars: Definitions, Systems, and Bibliography*, Lecture Notes in Computer Science, v. 323, Springer Verlag, 1988.

[Knuth 68] Donald E. Knuth, *Semantics of Context-Free Languages* Mathematical Systems Theory, Vol. 2, No. 2, pp. 127-145, 1968.

Index