Ox:

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

Tutorial Introduction

November 5, 1993

bischoff@cs.iastate.edu

1 Introduction

Ox is an attribute-grammar compiling system based on Yacc, Lex, and C. Ox [1] generalizes the function of Yacc in the way that attribute grammars generalize context-free grammars. Ordinary Yacc and Lex specifications can be augmented with definitions of synthesized and inherited attributes written in C/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 can specify parse-tree 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.

Ox is a Yacc/Lex/C/C++ preprocessor, and is designed to bring attribute grammars to the mainstream of Unix-based language development. Ox inherits all of the familiar syntax and semantics of Yacc, Lex, and C/C++. This makes Ox easily accessible to language designers, developers, and experimenters who use those tools. It also provides a ready "escape hatch" in case it is desired to return to an ordinary Yacc implementation.

This paper gives an overview of Ox by emphasizing examples. It quickly familiarizes you with the Ox features that are most immediately useful. A more complete reference, the *Ox User Reference Manual*, accompanies the Ox electronic distribution, which can be obtained free by writing to `ox-request@cs.iastate.edu`.

[1] The name "Ox" comes from an attempt to pronounce an acronym for "An Attribute-Grammar Compiling System"

Familiarity with the use of Yacc, Lex, C, and Make is sufficient to understand this tutorial and to begin using Ox. Some prior exposure to attribute grammars is helpful. Readers with an urge for details and hands-on experience should use the index of the reference manual and should have access to a system on which Ox is installed. The examples herein (in machine-readable form) are included with the Ox distribution.

2 Converting a Yacc/Lex program for use with Ox

Probably the easiest way to get started with Ox is to convert an existing Yacc/Lex [2] parser or translator. This can usually be done without changing the Yacc and Lex code. Ox can also be used with Yacc-only translators, i.e., those with lexical analyzers hand-coded in C (see section 9.3).

2.1 A parser of arithmetic expressions

As a running example, we start with a Yacc/Lex parser for integer arithmetic expressions.

The Lex file is named `scan.l`, and specifies the tokens of the language as digit strings, parentheses, and four binary operators:

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

%%
[ \n\t\f]+          ;
[0-9]+              return(ICONST);
[()*/+\-]           return(yytext[0]);
%%
```

The Yacc file (named `gram.y`) specifies the syntax. The grammar is disambiguated by use of the `%left` reserved word:

[2] Ox is designed to work also with Yacc and Lex workalikes and C++. Throughout this paper, "Yacc", "Lex", and "C" can generally be taken to mean "Yacc or Bison", "Lex or Flex", and "C or C++", respectively.

```
%token ICONST
%left '+' '-'
%left '*' '/'

%%
expr    :       expr    '*'     expr
        |       expr    '/'     expr
        |       expr    '+'     expr
        |       expr    '-'     expr
        |       '('     expr    ')'
        |       ICONST
        ;
%%
main()
  {return(yyparse());
  }
```

The following Make file is used to build and maintain the parser, which is named gc:

```
gc: y.tab.o lex.yy.o
        cc -o gc y.tab.o lex.yy.o -ly -ll

y.tab.c y.tab.h: gram.y
        yacc -d gram.y

lex.yy.c: scan.l
        lex scan.l

y.tab.o: y.tab.c
        cc -c y.tab.c

lex.yy.o: lex.yy.c y.tab.h
        cc -c lex.yy.c
```

2.2 A parser that builds a parse tree

The above parser does no semantic analysis. To get ready for Ox implementation of semantics, we need merely replace the following lines in the Make file:

```
y.tab.c y.tab.h: gram.y
        yacc -d gram.y

lex.yy.c: scan.l
        lex scan.l
```

with these:

```
oxout.y oxout.l: gram.y scan.l
        ox gram.y scan.l

y.tab.c y.tab.h: oxout.y
        yacc -d oxout.y

lex.yy.c: oxout.l
        lex oxout.l
```

The command:

```
ox gram.y scan.l
```

transforms `gram.y` (called the *Y-file*) into `oxout.y`, and transforms `scan.l` (called the *L-file*) into `oxout.l`. These Ox outputs replace `gram.y` and `scan.l` in the remaining steps of parser construction.

The user-observed behaviors of the original program and the one preprocessed by Ox are the same. The difference is that the version made using Ox and the new Make file builds a dummy (attribute-less) parse tree, while the original builds no parse tree. The original code in the example lacks Yacc actions, but had it contained such actions, their effects would have been undisturbed by the Ox preprocessing.

Having modified our Make file, we are ready to augment the Y-file and L-file with Ox constructs.

3 Adding Ox-generated semantics

This section introduces the form and meaning of Ox-specific constructs, by way of converting our parse-tree-building parser into a calculator.

Each parse tree has leaves labeled by the `ICONST` token. Let us endow this token with an attribute `string`: a character pointer that for each `ICONST` node is to point to a copy of the lexeme corresponding to the node. This is done by placing the *attribute declaration*:

```
@attributes {char *string;} ICONST
```

before the first `%%` mark in the Y-file. The above-mentioned storage location created for each `ICONST` node is called an *attribute instance* (concisely: an *instance*). It is an instance of the `string` attribute of `ICONST`.

We supply a C macro (named `lexeme`) that constructs a copy of the lexeme. For brevity of the example, `lexeme` unsafely neglects to check for return of `NULL` by `malloc`. Here is the modified L-file:

```
%{
#include "y.tab.h"
#include <string.h>

#define lexeme strcpy((char *)malloc(yyleng+1),yytext)
%}

%%
[ \n\t\f]+          ;
[0-9]+              return(ICONST); @{ @ICONST.string@ = lexeme; @}
"("                 return('(');
")"                 return(')');
"*"                 return('*');
"/"                 return('/');
"+"                 return('+');
"-"                 return('-');
%%
```

To the right of the lexical rule for `ICONST`, there is between `@{` and `@}` an *attribute definition* that causes the `string` attribute instance in each `ICONST` node to get a pointer to a copy of the constant's lexeme.

Notice that we have replaced the single lexical rule:

```
[()*/+\-]           return(yytext[0]);
```

with six rules that are together equivalent to that single rule. Ox would have been unable to determine from the object of the single `return` statement (namely `yytext[0]`) the specific token that would be `return`ed. By replacing the rule, we make the `return`ed tokens explicit, and avoid a warning from Ox.

Each parse-tree node labeled by `expr` is the root of a subtree corresponding to a subexpression. Placing the attribute declaration:

```
@attributes {long val;} expr
```

in the Y-file causes the Ox-generated translator to allocate space (an attribute instance) for a `long` named `val` each time it creates a node labeled by `expr`.

The *body* (the part between curly braces) of an attribute declaration resembles that of a C structure declaration, except that curly braces cannot be nested.[3]

The definitions for the `val` attribute of `expr` are seen in the *attribute reference sections* (code fragments delimited by `@{` and `@}`) in the modified Y-file. Each of the attribute definitions starts with the *implicit-mode annunciator* `@i`, whose meaning is explained in section 4.1. In this example, each attribute reference section contains exactly one attribute definition.

[3] Attributes can be of any C fundamental or derived type. The Ox code in section 8 uses an attribute that is a C structure.

```
%token ICONST
%left '+' '-'
%left '*' '/'

@attributes {char *string; } ICONST
@attributes {long val;}      expr

%%
expr    :       expr    '*'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ * @expr.2.val@;  @}

        |       expr    '/'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ / @expr.2.val@;  @}

        |       expr    '+'     expr
                @{ @i @expr.val@ = @expr.1.val@ + @expr.2.val@;    @}

        |       expr    '-'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ - @expr.2.val@;  @}

        |       '('     expr    ')'
                @{ @i @expr.0.val@ = @expr.1.val@;                 @}

        |       ICONST
                @{ @i @expr.val@ = atoi(@ICONST.string@);          @}
        ;
%%
main()
  {return(yyparse());
  }
```

The grammar symbol `expr` has three *grammar-symbol occurrences* (namely `expr.0`, `expr.1`, and `expr.2`) in the grammar rule:

```
expr    :       expr    '*'     expr
```

An *attribute occurrence* (concisely: an *occurrence*) is a grammar-symbol occurrence together with an attribute of the symbol. An *attribute reference* takes the form:

`@`*grammarsymbol*.[*integer*.]*attributename*`@`

where *attributename* appears as an identifier in the body of the attribute declaration for *grammarsymbol*. If integer is n, the reference is to the nth occurrence of *grammarsymbol* counting from the left of the rule (the leftmost occurrence being the 0th). The square brackets above denote that *integer* and the second `.` are optional (the default value for *integer* being 0).

Attribute definitions are basically C code fragments containing attribute references. In general, an attribute definition section contains zero to many attribute definitions. Each attribute definition is announced by a mode annunciator, and terminated by `@}` or by the next mode annunciator.

4 Order of Attribute-Instance Evaluation

Attribute grammars specify semantics in a *declarative* or *functional* (rather than sequential or imperative) style. When a parse tree is created, the tree's attribute instances are evaluated in an order constrained (but not fully determined) by the attribute grammar. It is clear that in the example of section 3, all `val` instances in the leaf nodes must be evaluated before the `val` instance of the root node.

Ox (rather than the compiler designer) generates code that causes instances to be evaluated in a correct order.

4.1 Dependency relations in the Y-file

There is a constraint for each grammar rule in the Y-file: a *dependency relation* on the attribute occurrences in that rule. For the rule:

```
expr    :       expr    '*'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ * @expr.2.val@;
                @}
```

there is the constraint that instances corresponding to `expr.1.val` and `expr.2.val` in sibling parse-tree nodes must be evaluated before the one corresponding to `expr.0.val` in their parent node.

Each rule's dependency relation is determined by its individual attribute definitions. There are several modes for communicating dependency information to Ox. The *implicit mode* is, for most Ox translators, the only such mode needed. The *explicit mode* is described briefly in section 9.7.

The implicit-mode annunciator `@i` (see the example in section 3) signals to Ox the beginning of an attribute definition. Further, it informs Ox that an instance corresponding to the definition's *leftmost* attribute reference is to be evaluated *after* those corresponding to other attribute references in the definition. This is to say that the occurrence corresponding to the leftmost reference *depends on* the occurrences corresponding to the other references in the definition.

4.2 Dependency relations in the L-file

Note that the mode annunciator `@i` does not appear in the L-file of the example in section 3. Mode annunciators are not used in L-files. An attribute reference section in an L-file is executed as a whole whenever the corresponding lexical rule is matched. In the example, this is done whenever the Lex-generated scanner matches a digit string. Executing an attribute reference section may involve the evaluation of several attribute instances. An attribute reference section in the L-file must contain exactly one attribute reference for each attribute occurrence defined there (in the previous example, that for `ICONST.string`).

5 Using global variables

Attribute reference sections can contain any C code, including references to global variables.

In our running example, we haven't yet shown how to print the main result of the semantic analysis (i.e., the value of the expression). The approach is to copy the `val` attribute instance of the root node into a global variable, then print it after termination of `yyparse()`. We introduce a unique start production for this purpose. The L-file need not be changed. Here is shown the new Y-file, with changed or added lines marked by empty C comments:

```
%token ICONST
%left '+' '-'
%left '*' '/'

%{                                                          /* */
long globVal;                                               /* */
%}                                                          /* */

@attributes {char *string;} ICONST
@attributes {long val;}     s expr                          /* */
%%
s       :       expr                                        /* */
                @{ @i globVal = @s.val@ = @expr.val@;      @}   /* */
        ;                                                   /* */

expr    :       expr    '*'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ * @expr.2.val@;  @}

        |       expr    '/'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ / @expr.2.val@;  @}

        |       expr    '+'     expr
                @{ @i @expr.val@ = @expr.1.val@ + @expr.2.val@;    @}

        |       expr    '-'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ - @expr.2.val@;  @}

        |       '('     expr    ')'
                @{ @i @expr.0.val@ = @expr.1.val@;                 @}

        |       ICONST
                @{ @i @expr.val@ = atoi(@ICONST.string@);          @}
        ;
%%
main()
  {yyparse();                                               /* */
   printf("%d\n",globVal);                                  /* */
  }
```

Upon completion of the call to `yyparse`, the tree's attribute instances have all been evaluated. The evaluation of `@s.val@` entails an assignment to `globVal`. The printing of `globVal` is the last thing done by the calculator.

6 Parse-tree traversals

A parse tree is much more useful if it can be traversed, and if its attribute instances can be accessed during traversals. Such traversals are particularly useful for code generation. Ox can be instructed to generate a translator that performs various kinds of traversals after evaluation of all of the tree's attribute instances.

6.1 Application: translation to prefix

The following Y-file specifies an expression parser that translates its (infix) input to prefix form. The L-file is the same as that of the previous example.

```
%token ICONST
%left '+' '-'
%left '*' '/'

@traversal @preorder yourTrav
@traversal @preorder yoursToo

@attributes {char *string;} ICONST
@attributes {long val;}     s expr
%%
s       :       expr
                @{ @i @s.val@ = @expr.val@;
                   @yoursToo printf("\n%d\n",@s.val@);
                @}
        ;
expr    :       expr    '*'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ * @expr.2.val@;
                   @yourTrav printf(" * ");
                @}
        |       expr    '/'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ / @expr.2.val@;
                   @yourTrav printf(" / ");
                @}
        |       expr    '+'     expr
                @{ @i @expr.val@ = @expr.1.val@ + @expr.2.val@;
                   @yourTrav printf(" + ");
                @}
        |       expr    '-'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ - @expr.2.val@;
                   @yourTrav printf(" - ");
                @}
        |       '('     expr    ')'
                @{ @i @expr.0.val@ = @expr.1.val@;
                @}
        |       ICONST
                @{ @i @expr.val@ = atoi(@ICONST.string@);
                   @yourTrav printf(" %s ",@ICONST.string@);
                @}
        ;
%%
main()
  {return(yyparse());
  }
```

The line

```
@traversal @preorder yourTrav
```

declares a left-to-right preorder traversal named `yourTrav`. Suppose that in our example, the `yourTrav` traversal has reached a node at which a grammar rule R is applied. If the attribute reference section of R contains the *traversal-mode annunciator* `@yourTrav` (which was given meaning by its `@traversal` declaration), then the `printf` statment following `@yourTrav` is executed, and the traversal is continued for the subtree rooted at the node in question. Using `@postorder` instead of `@preorder` would cause a traversal that executes the `printf` *after* completing the traversal of that subtree, resulting in a postfix translation.

A traversal that accesses the `val` instance in the root node is an alternative to using the global variable `globVal` of section 5. Placing the line:

```
@traversal @preorder yoursToo
```

in the declarations section, and the line:

```
@yoursToo printf("%d\n",@s.val@);
```

in the attribute reference section for the start production accomplishes the same thing as the use of `globVal`.

One traversal is done for each traversal declaration, the traversals being done one after another, in the order in which the declarations appear. In the example, the declaration of `yoursToo` appears after that of `yourTrav`, so the value of the expression is printed after the preorder translation is printed.

7 Inherited vs. Synthesized Attributes

It is useful to think of the lexical rules (i.e., the rules in the L-file) as virtual grammar rules (productions) whose right-hand sides are the empty string and whose left-hand sides, while actual Yacc tokens, are virtual nonterminals. This generic concept of *rule* is consistent with usual concepts of *attribute grammar*, and leads to the following definitions:

An attribute occurrence o in a rule R is *synthesized* if and only if

- o is on the LHS of R and the attribute reference section of R contains a definition of o, or
- 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

- o is on the left-hand side (LHS) of R and the attribute reference section of R contains no definition of o, or
- o is on the right-hand side (RHS) of R and the attribute reference section of R contains a definition of o.

Ox issues an error message if it finds an attribute that has 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 can 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) can have both synthesized and inherited attributes. Each symbol has a distinct name space, so same-named attributes of different symbols are distinct attributes, and can 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.

8 Using inherited attributes

This section gives an example indicating the use of inherited attributes for semantic analysis involving right context. The example also gives a better idea of how Ox code is used together with C code.

In many languages, for instance Pascal, each variable declaration is essentially a list of identifiers followed by a type specifier. Here we show a simple language whose every sentence consists of such a variable declaration. Our translator parses the input, recording in a symbol object the identifier and type of each variable declared. Then the symbol objects are printed during a postorder traversal.

Here is the L-file:

```
%{
#include "y.tab.h"
#include <string.h>

#define lexeme strcpy((char *)malloc(yyleng+1),yytext)
%}

%%
[ \n\t\f]+          ;
real                return(REAL);
integer             return(INT);
boolean             return(BOOL);
[a-zA-Z]+           return(IDENT);  @{ @IDENT.string@ = lexeme; @}
","                 return(',');
";"                 return(';');
":"                 return(':');
.                   {fprintf(stderr,"illegal character\n"); exit(-1);}
%%
```

The definitions in section 7 together with the following Y-file imply that:

- `string` is a synthesized attribute of `IDENT`.
- `sym` is an inherited attribute of `IDENT`.
- `tMark` is an inherited attribute of `varList`.
- `varDecl` has no attributes.

```
%token REAL INT BOOL IDENT

%{
#include <stdlib.h>
struct sym {char *str,*typeMark;};

struct sym *allocSym(cp,t)
  char *cp,*t;
  {struct sym *pSym;
   pSym = (struct sym *) malloc(sizeof (struct sym));
   pSym->str = cp; pSym->typeMark = t;
   return pSym;
  }
%}

@attributes {char *string; struct sym *sym; } IDENT
@attributes {char *tMark;                   } varList
@traversal @postorder myT /* my Traversal */

%%
varDecl    :        varList ':' REAL ';'
                    @{ @i @varList.tMark@ = "real";    @}

           |        varList ':' INT ';'
                    @{ @i @varList.tMark@ = "integer"; @}

           |        varList ':' BOOL ';'
                    @{ @i @varList.tMark@ = "boolean"; @}
           ;

varList    :        IDENT
                    @{ @i @IDENT.sym@ =
                            allocSym(@IDENT.string@,@varList.tMark@);
                       @myT printf("%s: %s;\n",@IDENT.sym@->typeMark,
                                               @IDENT.sym@->str);
                    @}
           |        varList ',' IDENT
                    @{ @i @varList.1.tMark@ = @varList.tMark@;
                       @i @IDENT.sym@ =
                            allocSym(@IDENT.string@,@varList.tMark@);
                       @myT printf("%s: %s;\n",@IDENT.sym@->typeMark,
                                               @IDENT.sym@->str);
                    @}
           ;
```

```
%%
main()
  {return(yyparse()); }
```

9 Overview of other features

This section briefly describes some Ox features that are provided for convenience or for advanced or specialized use. Detailed descriptions of these features appear in the *Ox User Reference Manual*.

9.1 Macro facility

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.

9.2 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. Ox syntax provides ways of specifying that a copy rule is global to the attribute grammar, obviating repetition of attribute definitions in many grammar rules.

9.3 Using Ox with scanners not based on Lex

By default, Ox provides preprocessing for Lex files augmented with Ox constructs. By using a command line option, Ox can be informed that the L-file contains Ox-augmented C code rather than the usual Ox-augmented Lex code.

9.4 Use of multiple scanners

Some translators contain several scanners. Such a translator is designed so that at any moment, it is using one scanner or another, and switches to a different one when there is a change in context. An Ox translator that uses more than one scanner can be constructed by submitting to Ox more than one L-file.

9.5 Stripping Ox constructs

Occasionally, the Ox user may desire copies of the Y-file and L-file(s) stripped of Ox-specific constructs. By a command-line option, the Ox user can filter all Ox-specific constructs from the inputs, to obtain 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.

9.6 Accessing Yacc pseudovariables

Attribute definitions that refer to the Yacc pseudovariables `$$`, `$1`, `$2`, etc. are permitted in various forms, including:

`@i @`*grammarsymbol*`.`[*integer*`.`]*attributename*`@ = $`*n*;

where `$`*n* denotes a Yacc pseudovariable. It is also possible to copy attribute instances into pseudovariables.

9.7 Expressing dependencies explicitly

Suppose that you have a C function `fun` in a library, and that you want to use it to define an attribute occurrence, say `sym.attrb`, in terms of some other occurrence `othersym.otherAttrb`. Further suppose that the first formal parameter of `fun` is of the same type as `othersym.otherAttrb`, and that `fun`'s second formal parameter is a pointer to something of the same type as `sym.attrb`. A call to `fun` changes the contents of the location indicated by its second argument.

It wouldn't work to write:

```
@i fun(@otherSym.otherAttrb@, &@sym.attrb@);
```

since the mode annuciator `@i` (see section 4.1) implies that the occurrence appearing first (`otherSym.otherAttrb`) is the occurrence being defined, and that it depends on `sym.attrb`. Actually you intend the opposite.

One solution would be to modify the definition of `fun` (reversing the order of its formal parameter list). If you don't want to disturb the library, however, it would be best to use Ox's *explicit mode annunciator* `@e` as follows:

```
@e sym.attrb : otherSym.otherAttrb ;
   fun(@otherSym.otherAttrb@, &@sym.attrb@);
```

In the first line above, Ox is explicitly given dependency information using a Make-like syntax: it is declared that `sym.attrb` depends on `otherSym.otherAttrb`. Use of the explicit mode makes the order of the occurrences in the second line's call to `fun` irrelevant to Ox's understanding of the dependencies.

9.8 Generating ANSI/ISO/C++ output

By default, the C code generated by Ox follows traditional C syntax. There is, however, a command-line option to produce code compatible with ANSI/ISO/C++ syntax. Thus it is easy to make Ox's code generation conform to the expectations of practically any C or C++ compiler.

10 Acknowledgements

This is to thank Terry Dineen, Carolyn Giberson, Markus Klingspor, John Levine, Carla Marceau, and Michael Seager for their helpful reviews of early versions of this paper.