SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 25(7), 789–810(JULY 1995)

ANTLR: A Predicated-$LL(k)$ Parser Generator

T. J. PARR
University of Minnesota, AHPCRC, 1100 Washington Ave S Ste 101, Minneapolis, MN 55415, parrt@acm.org

AND

R. W. QUONG
School of Electrical Engineering, Purdue University, W. Lafayette, IN 47907, quong@ecn.purdue.edu

SUMMARY

Despite the parsing power of $LR/LALR$ algorithms, e.g. YACC [1], programmers often choose to write recursive-descent parsers by hand to obtain increased flexibility, better error handling, and ease of debugging. We introduce ANTLR, a public-domain parser generator that combines the flexibility of hand-coded parsing with the convenience of a parser generator, which is a component of PCCTS [2]. ANTLR has many features that make it easier to use than other language tools. Most important, ANTLR provides *predicates* which let the programmer systematically direct the parse via arbitrary expressions using semantic and syntactic context; in practice, the use of predicates eliminates the need to hand-tweak the ANTLR output, even for difficult parsing problems. ANTLR also integrates the description of lexical and syntactic analysis, accepts $LL(k)$ grammars for $k > 1$ with extended BNF notation, and can automatically generate abstract syntax trees.

ANTLR is widely used, with over 1000 registered industrial and academic users in 37 countries. It has been ported to many popular systems such as the PC, Macintosh, and a variety of UNIX platforms; a commercial C++ front-end has been developed as a result of one of our industrial collaborations.

KEY WORDS Parsing Compiler Parser generator Predicates LL(k) parser

INTRODUCTION

Programmers want to use tools that employ mechanisms they understand, that are sufficiently powerful to solve their problem, that are flexible, that automate tedious tasks, and that generate output that is easily folded into their application. Consider parser generators. Existing parser generators often fail one or more of these criteria. Consequently, parsers are often written by hand, especially for languages that are context-sensitive or require large amounts of lookahead. Compared to a hand-built recursive-descent parser, table-driven LR/LL parsers often do not have enough parsing strength and can be difficult to understand and debug.

A parser must do much more than just recognize languages. In particular, parsers must interact with the lexical analyzer (scanner), report parsing errors, construct abstract syntax trees, and call user actions. Existing parsing tools have focused mainly on the language recognition strategy, often ignoring the aforementioned tasks.

In this paper, we introduce the ANTLR (ANother Tool for Language Recognition) parser generator, which addresses all these issues. ANTLR is a component of the Purdue Compiler Construction Tool Set (PCCTS)[2]. It constructs human-readable recursive-descent parsers in

0038–0644/95/070789–22$16.00 Received 10 June 1994

 Revised 17 November 1994

C or C++ from $pred\text{-}LL(k)$ [3,4] grammars, namely $LL(k)$ grammars, for $k > 1$, that support predicates.

Predicates allow arbitrary semantic and syntactic context to direct the parse in a systematic way. As a result, ANTLR can generate parsers for many context-sensitive languages and many non-$LL(k)$/$LR(k)$ context-free languages. Semantic predicates indicate the semantic validity of applying a production; syntactic predicates are grammar fragments that describe a syntactic context that must be satisfied before recognizing an associated production. In practice, many ANTLR users report that developing a pred-$LL(k)$ grammar is easier than developing the corresponding $LR(1)$ grammar.

In addition to a strong parsing strategy, ANTLR has many features that make it more programmer-friendly than the majority of $LR/LALR$ and LL parser generators.

- ANTLR integrates the specification of lexical and syntactic analysis. A separate lexical specification is unnecessary as lexical regular expressions (token descriptions) can be placed in double-quotes and used as normal token references in an ANTLR grammar.
- ANTLR accepts grammar constructs in Extended Backus-Naur Form (EBNF) notation.
- ANTLR provides facilities for automatic abstract syntax tree construction.
- ANTLR generates recursive-descent parsers in C/C++ so that there is a clear correspondence between the grammar specification and the ANTLR output. Consequently, it is relatively easy for non-parsing experts to design and debug an ANTLR grammar.
- ANTLR has both automatic and manual facilities for error recovery and reporting. The automatic mechanism is simple and effective for many parsing situations; the manual mechanism called "parser exception handling" simplifies development of high-quality error handling.
- ANTLR allows each grammar rule to have parameters and return values, facilitating attribute passing during the parse. Because ANTLR converts each rule to a C/C++ function in a recursive descent parser, a rule parameter is simply a function parameter. Additionally, ANTLR rules can have multiple return values.
- ANTLR has numerous other features that make it a product rather than a research project. ANTLR itself is written in highly portable C; its output can be debugged with existing source-level debuggers and is easily integrated into programmers' applications.

Ultimately, the true test of a language tool's usefulness lies with the vast industrial programmer community. ANTLR is widely used in the commercial and academic communities. More than 1000 registered users in 37 countries have acquired the software since the original 1.00 release in 1992. Several universities currently teach courses with ANTLR. Many commercial programmers use ANTLR; we list some examples in the Appendix.

For example, a major corporation [5] has nearly completed and is testing a unified C/Objective-C/C++ compiler using an ANTLR grammar that was derived directly from the June 1993 ANSI X3J16 C++ grammar. [Preliminary measurements show that this ANTLR parser is about 20% slower, in terms of pure parsing speed, than a hand-built recursive-descent parser that parses only C/Objective-C, but not C++]. C++ has been traditionally difficult for other $LL(1)$ tools and $LR(1)$-based tools such as YACC [1]. YACC grammars for C++ are extremely fragile with regards to action placement; i.e., the insertion of an action can introduce conflicts into the C++ grammar. In contrast, ANTLR grammars are insensitive to action placement due to their $LL(k)$ nature.

The following sections illustrate ANTLR's specification language and the features that distinguish it from other parser generators. As this paper is an overview, we have omitted many details. Refer to the current PCCTS/ANTLR release notes for complete usage details.

Table I. ANTLR description elements.

<table>
<tr><th>Item</th><th>Description</th><th>Example</th></tr>
<tr><td>Token</td><td>begins with uppercase letter</td><td><code>ID</code></td></tr>
<tr><td>Tokclass</td><td>set of tokens (token class)</td><td><code>Operators</code></td></tr>
<tr><td>~Tokclass</td><td>complement set of Tokclass</td><td><code>~Keyword</code></td></tr>
<tr><td>.</td><td>wild card token class</td><td><code>/* match ID or something else */</code>
<code>a : ID | . ;</code></td></tr>
<tr><td>rule_name</td><td>begins with lowercase letter</td><td><code>expr</code></td></tr>
<tr><td>label:item</td><td>label token or rule reference</td><td><code>a : t:ID ;</code></td></tr>
<tr><td><code><<...>></code></td><td>user-defined semantic action</td><td><code><<printf("%s", t->name);>></code></td></tr>
<tr><td><code>(...)</code></td><td>subrule</td><td><code>("int" | ID | storage_class)</code></td></tr>
<tr><td><code>(...)*</code></td><td>closure</td><td><code>ID ("," ID)*</code></td></tr>
<tr><td><code>(...)+</code></td><td>positive closure</td><td><code>slist : (stat | SEMICOLON)+ ;</code></td></tr>
<tr><td><code>{...}</code></td><td>optional</td><td><code>{ELSE stat }</code></td></tr>
<tr><td><code><<...>>?</code></td><td>semantic predicate</td><td><code>type : <<is_TYPE(str)>>? ID ;</code></td></tr>
<tr><td><code>(...)?</code></td><td>syntactic predicate</td><td><code>((list EQ)? list EQ list | list)</code></td></tr>
</table>

DESCRIPTION LANGUAGE

An ANTLR *description* or *specification* is a collection of rules and actions preceded by a header in which the user defines required data types, such as the type of an attribute. We originally borrowed the notation from YACC to reduce the learning curve, but since then, we have added numerous extensions for new ANTLR features such as predicates, specification of lexical analysis, error reporting, and EBNF grouping. Table I summarizes the elements in an ANTLR description.

An ANTLR rule is a list of productions or alternatives separated by "|":

$$
\begin{array}{ll}
\texttt{rule} & \texttt{:}\ alternative_1 \\
 & |\ alternative_2 \\
 & \quad\vdots \\
 & |\ alternative_n \\
 & \texttt{;}
\end{array}
$$

where each alternative production is composed of a list of elements; an element is an item from Table I. In this table, the "`...`" within the grouping constructs can themselves be lists of alternatives or items.

Rules may also define arguments and return values. In the following line, there are n arguments and m return values.

$$\texttt{rule[}arg_1\texttt{, ..., }arg_n\texttt{] > [}retval_1\texttt{, ..., }retval_m\texttt{] : ... ;}$$

The syntax for using a rule mirrors its definition,

$$
\begin{array}{lll}
\texttt{a} & \texttt{:} & \texttt{... rule[}arg_1\texttt{,}\ldots\texttt{,}arg_n\texttt{] > [}v_1\texttt{,}\ldots\texttt{,}v_m\texttt{] ...} \\
 & \texttt{;} &
\end{array}
$$

Here, the various v_i receive the return values from the rule `rule`, so that each v_i must be an *l-value*.

We illustrate the major features of ANTLR's description language via a small example. Consider parsing the following simple assembly language.

```
#segment data
a  ds 42
b  ds 13
#segment code
   load r1, a
   load r2, b
   add  r1,r2,r3
   print r3
```

Figure 1 contains a complete ANTLR specification for this problem.

```
#header <<#include "charbuf.h">>

<<main() {ANTLR(prog(), stdin); }>>

#tokclass OPCODE   {"add" "store" "load" "call" "ret" "print" }
#tokclass REGISTER {"r0" "r1" "r2" "r3" }

#token "[\ \t]+"     <<zzskip();>>
#token "\n"          <<zzskip(); zzline++;>>

prog:   "#segment" "data" (data)*
        "#segment" "code" (stat)*
    ;
stat:   OPCODE operands
    ;
operands
    :   ID
    |   REGISTER
    |   REGISTER "," NUM
    |   REGISTER "," REGISTER "," REGISTER
    ;
data:   ID "ds" NUM
    ;
#token NUM     "[0-9]+"
#token ID      "[a-zA-Z]+"
```

Figure 1. ANTLR recognizer for a simple assembly language.

A description for ANTLR differs from those of other parser generators because regular expressions specifying tokens are specified (`#token ID`) or directly referenced (`#segment`) in the grammar. Thus, both the grammatical and the lexical specification are contained in a single file, eliminating the need to maintain two specifications. ANTLR automatically assigns token types and generates a scanner description for DLG, the lexical analyzer generator in PCCTS. Regular expression ambiguities, such as between keyword `code` and token `ID`, as the input "code" could be either token type, are resolved by matching the token specification mentioned first in the ANTLR grammar.

ANTLR accepts input in EBNF, as shown in Table I, which simplifies grammar development and grammar readability. (Strict BNF accepts neither subrules, closures, nor optional items in a grammar specification.) As a further notational shorthand, ANTLR accepts arbitrary sets of tokens called *token classes*. The user specifies a token class as a set of tokens or other token classes. A token class is functionally equivalent to a subrule whose alternatives are its member tokens; e.g., referencing token class REGISTER in Figure 1 is the same as referencing

```
( "r0" | "r1" | "r2" | "r3" )
```

Using a token class is more efficient than using a subrule, because referencing a token class is a simple set membership test. In contrast, referencing the equivalent subrule causes a sequential search of its alternative items because subrule items might be predicates or other rules. The code to test for set membership is much smaller than a series of if-else statements for a subrule. Note that automaton-based parsers (both LR and LL) automatically perform this type of set membership (specifically, a table lookup), but lack the flexibility of recursive-descent parsers; e.g., recursive-descent parsers directly support programmer-defined stack-based local-variables.

Note that the grammar in Figure 1 is not $LL(1)$ as it is not left-factored. Because ANTLR generates $LL(k)$ recursive-descent parsers, with $k > 1$, ANTLR grammars require less left-factoring than $LL(1)$ grammars. The grammar in Figure 1 is $LL(3)$, as three symbols of lookahead suffice to differentiate between the alternatives of operands when at the left edge of operands. In fact, this grammar also contains decisions that require lookahead of one symbol (subrule (data)*), and two symbols (rule prog).

ANTLR optimizes lookahead decisions by using as little lookahead as possible, even within the same decision. For example, although rule operands requires three lookahead symbols to distinguish between the last two alternatives, ANTLR generates a decision that uses only one lookahead symbol to distinguish the first alternative from the other three. Thus, the programmer can use the power of $k > 1$ lookahead without worrying about efficiency.

User-defined actions can be inserted anywhere in an ANTLR production. Such actions are often used to perform semantic tests, generate an intermediate representation, or directly generate a translation. An action placed at the beginning of the first production of any rule or subrule is special; these actions are *init-actions* and can be used to define local variables or execute code that must be executed before any production is attempted. Thus, an init-action applies to all productions in a rule. Local variables are useful for recursively-invoked rules because a new copy of a variable is available per rule invocation; in contrast, simulating local variables in a table-driven parser requires a software stack, which is inconvenient and tedious to implement.

Actions in an ANTLR grammar may access attributes via labels (attached to token and rule references) of the form *$label* rather than the conventional "$\$i$" for some integer i. By using symbols instead of integer identifiers, grammars are more readable and actions are not sensitive to positional changes to rule elements.

PARSING STRENGTH

$LL(k)$ Parsing For $k > 1$

ANTLR $pred\text{-}LL(k)$ parsers compare favorably to $LR(1)$ parsers [6]. In both theory and practice, there are languages that are $LR(1)$ but not $LL(3)$, and vice versa. Use of $k = 2$ or $k = 3$ significantly reduces the need to left-factor rules. In all cases, ANTLR computes and uses the minimum lookahead necessary for each decision within the grammar, speeding up grammar analysis and parsing speed and reducing the parser code size. In practice, one lookahead token suffices for many decisions, so ANTLR parsers are nearly as efficient as $LL(1)$ recursive-descent parsers.

We now illustrate how an $LL(2)$ grammar can be much simpler to design than an $LL(1)$ grammar. Consider distinguishing between C labels "`ID :`" and C assignment statements "`ID =` ..." when parsing. In the following grammar fragment, rule `stat` requires two lookahead symbols, and is easily expressed with an $LL(2)$ grammar. This common language feature is hard to express in an $LL(1)$ grammar, because `ID` is matched in many grammar locations making it difficult to left-factor rules `stat` and `expr`.

```
stat:   ID ":" stat          /* statement label */
    |   expr ";"             /* assignment stat */
    ;
expr:   ID "=" expr
    |   INT
    ;
```

Although using $k > 1$ symbols of lookahead is useful, there are many language constructs that are not $LL(k)$ for any finite k. Typically, these constructs are context sensitive or require unbounded lookahead (i.e., the entire construct must be seen before it may be uniquely identified). We outline the predicate mechanism that allows ANTLR to handle many nasty recognition problems in the next section.

Predicates

ANTLR supports the use of the *semantic* and *syntactic predicates*, which let the programmer indicate the semantic and syntactic validity of applying a production, allowing ANTLR to naturally handle many difficult parsing situations. Predicates are described fully elsewhere[4]. Here, we present two simple examples demonstrating their power.

The first example illustrates semantic predicates. Consider FORTRAN array references and function calls, which are syntactically identical, but semantically different. Depending on the type of `VAL`, the following expression could be either an array reference or a function call.

```
VAL(13,I)
```

One common solution to resolve this syntactic ambiguity is for the lexical analyzer to examine the symbol table and to return a different lookahead token type based on whether the input identifier `VAL` is a variable or a function. The grammar would then reference different token types, say, `FUNC` and `VAR`, and would be context-free. However, semantic predicates provide a more elegant and more general solution. The same expression that would normally be used to return different token types may be used to alter the normal $LL(k)$ parsing strategy by annotating the grammar:

```
expr : <<isvar(LATEXT(1))>>?  ID "\(" expr_list "\)" << array_ref_action>>
     | <<isfunc(LATEXT(1))>>? ID "\(" expr_list "\)" << fn_call_action>>
     ;
```

where `isvar(LATEXT(1))` and `isfunc(LATEXT(1))` are user-defined functions that examine the symbol table and return true if first lookahead symbol, `LATEXT(1)`, is a variable or a function, respectively.

A semantic predicate is a user-defined action that evaluates to either true (success) or false (failure) and, broadly speaking, indicates the semantic validity of continuing with the parse beyond the predicate. Semantic predicates are specified via "$<<\ predicate\ >>$?" and may be interspersed among the grammar elements on the right hand side of productions like normal actions.

We now show how syntactic predicates are used via example. Occasionally, the programmer will face a language construct that cannot be parsed with an $LR(k)$ or $LL(k)$ parser even with the help of semantic predicates. Often these constructs simply require unbounded lookahead, that is, with a finite lookahead buffer, the parser is unable to determine which of a set of alternative productions to predict. We turn to parsing C++ for a nasty example. Quoting from Ellis and Stroustrup [7],

> "There is an ambiguity in the grammar involving *expression-statements* and *declarations* …The general cases cannot be resolved without backtracking …In particular, the lookahead needed to disambiguate this case is not limited."

The authors use the following examples to make their point, where `T` represents a type:

```
T(*a)->m=7;        // expression-statement with type cast to T
T(*a)(int);        // pointer to function declaration
```

Clearly, the two types of statements are not distinguishable from the left as an arbitrary number of symbols may be seen before a decision can be made; here, the "->" symbol is the first indication that the first example is a statement. Quoting Ellis and Stroustrup further,

> "In a parser with backtracking the disambiguating rule can be stated simply:
>
> 1. If it looks like a *declaration*, it is; otherwise
> 2. if it looks like an *expression*, it is; otherwise
> 3. it is a syntax error."

The solution in ANTLR is to use a syntactic predicate and simply to do exactly what Ellis and Stroustrup indicate,

```
stat:   (declaration)? declaration
    |   expression
    ;
```

In the first production of rule `stat`, the syntactic predicate `(declaration)?` indicates that `declaration` is the syntactic context that must be present for the rest of that production to succeed. We can interpret the use of "`(declaration)?`" as "I am not sure if `declaration` will match; let me try it out and, if it does not match, I shall try the next alternative." Thus, when encountering a valid declaration, the rule `declaration` will be recognized twice–once as syntactic predicate and once during the actual parse to execute semantic actions. If an expression is found instead, the declaration rule will be attempted at most once (`declaration` will not be attempted for obvious expressions such as "`a=3+4`").

Syntactic predicates have the form "`(` α `)?`" and may appear on the left edge of any production of a rule or subrule. The required syntactic condition, α, may be any valid context-free grammar fragment. Syntactic predicates were introduced into ANTLR version 1.10 [8]; they represent a form of selective backtracking that significantly enhances the recognition strength of normal $LL(k)$ parsing while not significantly increasing the parse time. (In our example, we parse local variable declarations twice, but these declarations typically are comprised of only a few symbols such as "`int i;`" and furthermore, `declaration` would not be attempted for obvious statements like `if`-statements because of the normal finite lookahead prediction).

Attribute parsing

A top-down parser can pass information into rules (*attribute inheritance*) as well as out of rules, namely it can perform L-attributed translations [9,10]; ANTLR is no exception. We illustrate the attribute passing facilities of ANTLR via a simple example. Consider a rule that recognizes declarations for both variable and function parameters. To distinguish between variables and parameters, we pass the current scope or context into the rule.

```
<<enum ScopeType {GLOBAL, PARAMETER};>>
globals
     :    ( declaration[GLOBAL] )*
     ;
func:     type ID "\(" ( declaration[PARAMETER] )* "\)"
     ;
declaration[ScopeType context]
     :    type ID << define variable based upon $context;>>
     ;
```

We have adapted the attribute-access notation, in which "`$context`" represents the value of the attribute passed into `declaration` from YACC. An attribute can be any valid C or C++ type.

HIGH-LEVEL PROGRAMMER SUPPORT

ANTLR contains a number of features that significantly increase its usability. In this section, we describe ANTLR's integrated lexical and syntactic descriptions, its error reporting facility, and its automatic tree construction mechanism.

Integration of lexical and syntactic analysis

An ANTLR description contains both the lexical-analyzer specification (for tokens) and the parser specification (the grammar), which eliminates the need to have separate files for each. ANTLR automatically extracts a lexical analyzer description from the integrated ANTLR description and passes it to DLG, the lexical analyzer generator in PCCTS.

```
#token INT  "[0-9]+"
stat:   "while" expr "do" stat
    |   "return" expr ";"
    ;
expr:  INT ( "\+" INT )*  /* match '+'-separated integers */
    ;
```

Tokens are declared either by a `#token` declaration or by direct reference in the grammar; the latter method is especially convenient for keywords. Tokens are specified as regular expressions, which must be enclosed in double quotes. For example, the preceding grammar declares the token `INT` via `#token`, and it directly refers to the `while` keyword as a token. We have labeled the specification for `INT` for clarity as it is used more than once. Consequently, DLG would receive six token specifications, one for each double-quoted regular expression.

As with other lexer generators, actions can be attached to token specifications. An action is executed when the corresponding token is recognized in the input stream. For example, the `#token` specification (using the C interface)

```
#token "\n" <<zzline++; zzskip();>>
```

indicates that when we encounter a newline character (`"\n"`), we increment the predefined line number variable `zzline` and then call the lexical analyzer to find another token, so that the parser need not see newlines.

ANTLR allows the use of multiple lexical analyzers within the same ANTLR description; this ability can simplify parsing of languages with wildly varying input formats. While other tools such as LEX allow multiple lexer automata within one description, the programmer is required to switch automata in lexical actions. This process is difficult without knowing the grammatical context, and is, therefore, much easier to do within a parser action.

In future versions of ANTLR, we anticipate allowing $pred$-$LL(k)$ constructs to describe input tokens. This idea from YACC++* would provide a consistent and powerful description language.

Error handling

ANTLR has two mechanisms for error reporting and recovery. In the first mechanism, ANTLR automatically generates error messages using a simple, effective heuristic that is sufficient for many applications. However, when more sophisticated error handling is required, say for commercial-quality software, ANTLR provides a second mechanism called *parser exception handling* that provides the flexibility of hand-built reporting and recovery in a convenient framework. We begin by describing ANTLR's automatic mechanism.

The automatic error handler reports where the error was detected and what was expected (recovery is discussed below). For example, consider matching the rule `stat` using the following grammar fragment,

* YACC++ is a registered trademark of Compiler Resources, Inc.

```
stat:   "if" expr "then" stat  "else" stat
    |   "while" expr "do" stat
    |   VAR ":=" expr ";"
    |   "begin" ( stat )+ "end"
    ;

expr:   atom ( "\+" atom )*
    ;

atom:   INT
    |   FLOAT
    ;
```

where `INT`, `FLOAT`, and `VAR` are defined as integer, float, and identifier tokens, respectively. Given the input

```
34
```

ANTLR automatically generates the error message

```
line 1: syntax error at "34" missing {if while VAR begin }
```

which indicates the first token of the syntax error and the set of tokens that would have been permissible.

Upon reading the input

```
if 34+ then i:=1;
```

the error message would be

```
line 1: syntax error at "then" missing {INT FLOAT}.
```

While correct, these messages could be clearer. Consequently, the user can specify *error classes*, which are named sets of tokens, so that ANTLR will report a more meaningful string in its default error messages. For example, after adding

```
#errclass Statement {"if" "while" VAR "begin" }
```

to the above grammar, the input of "`34`" would result in the error message

```
line 1: syntax error at "34" missing Statement.
```

The description of an error class EC consists of tokens, other error classes, and even rules. If EC contains rule r, we add the $FIRST$ set of r to $EC^{\dagger}$. This feature is convenient; for example, we can also specify the error class `Statement` via

```
#errclass Statement { stat }.
```

† A token t is in $FIRST$ of rule r if r might start with a t.

To recover after a parsing error in rule r, ANTLR consumes tokens until a token in the $FOLLOW$ set of r is found[‡]. This simple recovery heuristic works well in many cases. For example, after reporting the error message due to an incomplete `expr`, "`34 +`", on the input

```
if 34+ then i:=1;
```

the parser would look for a token that could follow an `expr`. Because "`then`" can follow an `expr`, the resynchronizer need not consume any tokens. Except for the error message, the parser returns from `expr` as if nothing had gone amiss and continues parsing the `then`-part of the `if` statement.

If the above default error mechanism is insufficient, programmers can use a more sophisticated error mechanism called *parser exception handling*, which has much in common with C++ exception handling [7]; we do not actually use C++ exceptions in our implementation and, hence, parser exception handling can be used with either the ANTLR C or C++ interface. Parser exception handling provides a unified framework for reporting and recovering from semantic and syntactic errors; note that automatic mechanisms typically do not even consider semantic errors. Parser exception handling provides nearly the flexibility of a hand-built parser.

We illustrate the use of parser exception handlers by demonstrating how they are used to generate a better error message than:

```
line 1: syntax error at "then" missing {INT FLOAT }.
```

for input:

```
if 34+ then i:=1;
```

Because we know the context in which the `expr` production was attempted, an improved error message would indicate the expression was both in an `if`-statement and that it was a conditional—as opposed to the right-hand-side of an assignment statement, for example. A better message would be

```
line 1: if-statement: malformed conditional at "then"
```

One way to achieve this error message is to modify the original `stat` grammar as follows

‡ A token t is in $FOLLOW$ of rule r if rule r can be followed immediately by a t.

```
stat: "if" e:expr "then" stat {"else" stat }
      exception[e]
        catch MismatchedToken :
        catch NoViableAlt :
           <<
           fprintf(stderr,
             "line %d: if-statement: malformed conditional at \"%s\"\n",
             zzline, LATEXT(1));
           zzconsumeUntilToken(THEN);
           >>
     | "while" expr "do" stat
       ⋮
     ;
```

where `zzline` is the current line number, `LATEXT(1)` is the text of the first token of lookahead (using the C interface), and `THEN` is the token type associated with `"then"`. The notation "`e:expr`" attaches the label `e` to the `expr` rule reference. Labels allows the exception handler to catch errors encountered specifically during that reference.

Good error handling requires programmer intervention. Automatic mechanisms typically do not perform well, because they cannot easily analyze the state of the parser (e.g., the symbol stack of a table-driven parser or the program counter of a recursive-descent parser). Knowing where to report errors and how to recover from them must be done with a programmer's experience. While more programming effort is required than for automatic mechanisms, ANTLR's parser exception handling provides a convenient, sophisticated mechanism that rivals the flexibility of hand-coded schemes.

Tree construction

The parser often constructs an intermediate form that is to be manipulated by later phases of the translation or compilation process. Using a few simple grammar annotations, ANTLR parsers can automatically construct abstract syntax trees (AST), saving the user from having to explicitly call tree constructor routines. Nodes in the AST are linked via left-most child and next-sibling pointers.

To create an AST, the user annotates the grammar to indicate what is a root node, what is a leaf node, and what is to be excluded from the AST. Tokens in the grammar immediately followed by "`^`" are to be considered subtree root nodes. Tokens suffixed with "`!`" are to be excluded from the tree. All other tokens are considered leaf nodes. For example, using the ANTLR specification in Figure 3 on the input

```
if 3+4*5 then return 4;
```

we would get the tree in Figure 2. The root of this tree would be returned as `root` in `main()`.

GENERATED PARSERS

ANTLR generates either C or C++ code for a recursive-descent parser, in which each grammar rule is realized by a C or C++ function. We illustrate the structure of these functions by example. For more information about the C/C++ output and the programmer's interface, refer

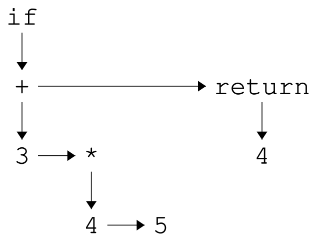

Figure 2. The abstract syntax tree resulting from “`if 3+4*5 then return 4;`”. *We use the left-most-child and next-sibling links.*

to the PCCTS release notes.

Each ANTLR-generated function is a sequence of `if-then-else` clauses plus an error clause. Each `if-then-else` clause matches one alternative of the corresponding rule; the `if` condition is a *prediction expression* for determining the validity of its alternative. For example, the rule `stat` from the following grammar

```
#header <<
    #include "charbuf.h"
    #define AST_FIELDS   int token, ival;
>>
<<
/* required function: how to convert from attribute to AST node */
void
zzcr_ast(AST *node, Attrib *cur, int token, char *text)
{
    node->token = token;
    node->ival = atoi(text);
}
main()
{
    AST *root=NULL;
    ANTLR(e(&root), stdin);
}
>>

stat:   "if"^ e "then"! stat ";"!
    |   "return"^ e
    ;
e   :   e1 ("\+"^ e1)* ;
e1  :   e2 ("\*"^ e2)* ;
e2  :   "[0-9]+" ;
```

Figure 3. ANTLR grammar showing AST construction directives using C interface.

```
stat:   ID COLON stat          /* statement label */
     |  expr SEMICOLON         /* assignment stat */
     |  RETURN expr
     ;
expr:   ID ASSIGN expr
     |  INT
     ;
```

would result in the following slightly-sanitized C code:

```
void stat(void)
{
  zzRULE;    zzBLOCK(zztasp1);    zzMake0;
  if ( (LA(1)==ID) && (LA(2)==COLON) ) {
    zzmatch(ID); zzCONSUME;
    zzmatch(COLON); zzCONSUME;
    stat();
  } else {
    if ( (LA(1)==ID||LA(1)==INT) && (LA(2)==SEMICOLON||LA(2)==ASSIGN) ) {
      expr();
      zzmatch(SEMICOLON); zzCONSUME;
    } else {
      if ( (LA(1)==RETURN) ) {
        zzmatch(RETURN); zzCONSUME;
        expr();
      } else
        error-clause;
    }
  }
  zzEXIT(zztasp1);
  return;
fail:              /* standard error-case code */
  ⋮
}
```

where `LA(`i`)` is the token type of the i^{th} symbol of lookahead; the terms `zzRULE`, `zzBLOCK`, `zzMake0`, and `zzEXIT` are bookkeeping macros for attribute manipulation. (Note that we have refrained from specifying lexical regular expressions, using token type labels instead, so that symbols appear in the C output rather than integer token types.)

Note that ANTLR adjusts the amount of lookahead tested even within the same parsing decision in an effort to reduce grammar analysis time and the size of the resulting parser. Thus, prediction expressions examine as few lookahead symbols as possible. In the above example, two lookahead symbols must be examined to distinguish between the first two alternatives, "`ID COLON stat`" and "`expr SEMICOLON`", as both can start with an `ID`, but only one lookahead symbol, `RETURN`, is tested for the third alternative because the `RETURN` token alone distinguishes it from the other two productions.

For efficiency, we considered the use of `switch`-statements rather than a sequence of `if-then-else`s, but `switch`es turned out to be too restrictive. For example, `switch`es cannot be used when $k > 1$ or when predicates are needed in the prediction expression. Also, parsing speed has not been a problem for ANTLR-generated parsers.

When semantic predicates are needed to disambiguate two or more alternative productions, we add the predicate to the prediction expression after the lookahead membership expression. For example, the grammar fragment

```
expr : <<isvar(LATEXT(1))>>?  ID "\(" expr_list "\)" << array_ref_action>>
     | <<isfunc(LATEXT(1))>>? ID "\(" expr_list "\)" << fn_call_action>>
     ;
```

would result in the following C code (again, we have lightly sanitized the code for clarity):

```
void expr(void)
{
    zzRULE;    zzBLOCK(zztasp1);    zzMake0;
    if ( LA(1)==ID && isvar(LATEXT(1)) ) {
        zzmatch(ID); zzCONSUME;
        zzmatch(3); zzCONSUME;            /* token type 3 refers to "(" */
        expr_list();
        zzmatch(4);                       /* token type 4 refers to ")" */
        array_ref_action
        zzCONSUME;
    } else {
        if ( LA(1)==ID && isfunc(LATEXT(1)) ) {
            zzmatch(ID); zzCONSUME;
            zzmatch(3); zzCONSUME;        /* token type 3 refers to "(" */
            expr_list();
            zzmatch(4);                   /* token type 4 refers to ")" */
            fn_call_action
            zzCONSUME;
        } else
            error-clause;
    }
    zzEXIT(zztasp1);
    return;
fail:                                     /* standard error-case code */
    ⋮
}
```

Implementing syntactic predicates is not as simple as implementing semantic predicates, because of the backtracking involved. For example, the rule

```
stat:   (declaration)? declaration
     |  expression
     ;
```

would result in the following C code

```
void stat(void)
{
    zzRULE;    zzBLOCK(zztasp1);    zzMake0;
    zzGUESS_BLOCK
    zzGUESS
    if ( !zzrv && (LA(1)==FIRST_1(declaration)) ) {
        {
            zzBLOCK(zztasp2);
            zzMake0;
            {
                declaration();       /* syntactic predicate */
                zzEXIT(zztasp2);
            }
        }
        zzGUESS_DONE
        declaration();
    } else {
        if ( zzguessing ) zzGUESS_DONE;
        if ( (LA(1)==FIRST_1(expression)) ) {
            expression();
        } else
            error-clause;
    }
    zzEXIT(zztasp1);
    return;
fail:                                /* standard error-case code */
    ⋮
}
```

where zzGUESS, and zzGUESS_DONE are bookkeeping macros to handle the backtracking.

Before evaluating a syntactic predicate, the state of the run-time stack is saved so that in case the predicate fails, a longjmp() can be used to restore the parser to its prior state before it attempted the predicate. Actions are not executed during the evaluation of a syntactic predicate to avoid side effects. If the predicate succeeds, parsing continues at the production predicated by the syntactic predicate, without executing the longjmp().

In this particular example, the syntactic predicate "(declaration)?" verifies that input will indeed match a declaration. We have used declaration to predict itself. Thus declaration will be matched twice—once as the syntactic predicate and then again during the actual parse to perform the actions specified in declaration.

C++ parsers

When generating recursive-descent parsers in C++, ANTLR uses the flexibility of C++ classes in two ways to create modular, reusable code. First, ANTLR will generate parser classes in which the class member functions, rather than global functions, contain the code (i) to recognize rules and (ii) to perform semantic actions. Second, ANTLR uses snap-together classes for the input, the lexer, and the token buffer.

An ANTLR parser consists of one or more C++ classes, called parser classes. Each parser class recognizes and translates part (or all) of a language. The recursive-descent recognition routines and the semantic actions are member functions of this class. A parser object is an instantiation (or variable) of the parser class.

To specify the name of the parser class in an ANTLR grammar description, enclose the appropriate rules and actions in a C++ class definition, as follows.

```
class Expr {
<<int i;>>
<<
public:
    void print();
>>
e   :   INT ("\*" INT)* ;
    ⋮                   // other grammar rules
}
```

ANTLR would generate a parser class `Expr` that looks as follows. The types `TokenType` and `ANTLRTokenBuffer` are discussed below.

```
class Expr : public ANTLRParser {
public:
    Expr(ANTLRTokenBuffer *input);
    Expr(ANTLRTokenBuffer *input, TokenType eof);
    void e();
    int i;
    void print();
private:
    internal- Expr-specific-data;
};
```

It is natural to have many separate parser objects. For example, if parsing ANSI C code, we might have three parser classes for C expressions, C declarations, and C statements. Parsing multiple languages or parts of languages simply involves switching parsers objects. For example, if you had a working C language front-end for a compiler, to evaluate C expressions in a debugger, just use the parser object for C expressions (and modify the semantic actions via virtual functions as described below).

Using parser classes has the standard advantages of C++ classes involving namespaces and encapsulation of state. Because all routines are class member functions, they belong in the class namespace and do not clutter the global namespace, reducing (or greatly simplifying) the problem of name clashes. The programmer can also specify some rules as `public`, such as the start rule, and make the rest of the rules `private`, clearly indicating which rules are for general use. Lastly, a parser object encapsulates the various state needed during a parse or translation.

While the ability to cleanly instantiate and invoke multiple parsers is useful, the main advantage of parser classes is that they can be extended in an object-oriented fashion. By using the inheritance and virtual functions mechanisms of C++, a parser class can be used as the base class (superclass) for a variety of similar but non-identical uses. Derived parser classes would be specialized for different activities; in many cases, these derived classes need only redefine translation actions, as they inherit the grammar rules, as these recursive-descent routines are member functions, from the base class.

As an example, assume that we have constructed an ANTLR specification for the front-end of a unified ANSI C/C++ compiler and that we use a parser class `StructClassParser` to handle C structs and C++ classes. The semantic actions `addDefn()` and `memberDefn()`, which add entries to the symbol table, would be defined as virtual member functions of `StructClassParser`.

```
class StructClassParser {
public:
<<
    virtual void addDefn(char *struct_className) {
        sym_tab->insert(struct_className);
            ⋮
    }
    virtual void memberDefn(char *memName) {
        sym_tab->insertMem(structName, memName);
            ⋮
    }
>>

s   : ("struct"|"class") name:ID << addDefn($name); >> "\{" (mem_decl)* "\}"
    ;

mem_decl
    : ...  <<  memberDefn(…);  >>
    ;
}
```

To create a unified C/C++ browser, we specify a second class `StructClassBrowser` derived from `StructClassParser` that simply redefines the semantic actions to be browser actions. In our sample code, the routine `display()` represents the action to show a string in the browser. As the recursive-descent recognition routines are member functions of base class `StructClassParser`, they are inherited by the browser class and we need not respecify the grammar. As the actions were virtual, the browser semantic actions will be called automatically when using a browser object.

```
class StructClassBrowser : public StructClassParser {
public:
    StructClassBrowser(ANTLRTokenBuffer *in) : StructClassParser(in) { }
    void addDefn(char *cl) {display("struct %s;\n", cl); }
    void memberDefn(char *m) {display("member \t%s;\n", m); }
};
```

The second way ANTLR uses C++ classes is to have separate C++ classes for the input stream, the lexical analyzer (scanner), the token buffer, and the parser. Conceptually, these classes fit together as shown in Figure 4, and in fact, the ANTLR-generated classes "snap together" in an identical fashion. To initialize the parser, the programmer simply

1. attaches an input stream object to a DLG-based scanner[§],
2. attaches a scanner to a token buffer object, and
3. attaches the token buffer to a parser object generated by ANTLR.

[§] If the user has constructed their own scanner, they would attach it here.

Figure 4. Overview of the C++ classes used by ANTLR.

The following code illustrates, for a parser object `Expr`, how these classes fit together.

```
main()
{
    DLGFileInput in(stdin);         // get an input stream for DLG
    DLGLexer scan(&in);             // connect a scanner to an input stream
    ANTLRTokenBuffer pipe(&scan);   // connect scanner and parser via pipe
    ANTLRToken aToken;
    scan.setToken(&aToken);         // DLG needs vtbl to access virtual fn
    Expr parser(&pipe);             // make a parser connected to the pipe
    parser.init();                  // initialize the parser
    parser.e();                     // begin parsing;  e = start symbol
}
```

`ANTLRToken` is defined by the programmer and must be a subclass of `ANTLRAbstractToken`. To start parsing, it is sufficient to call the `Expr` member function associated with the grammar rule; here, `e` is the start symbol.

To ensure compatibility among different input streams, lexers, token buffers, and parsers, all objects are derived from one of the four common bases classes `DLGInputStream`, `DLGLexer`, `ANTLRTokenBuffer` or `ANTLRParser`. In particular, all parsers are derived from a common base class `ANTLRParser`.

FUTURE WORK

Our work on ANTLR continues to be heavily influenced by the feedback from the industrial user community. As such, we are currently developing a prototype graphical user-interface that displays grammars as a set of syntax diagrams. This interface will highlight the conflicting syntax diagram paths in an invalid grammar construct, simplifying the debugging of a grammar. Currently, ambiguities are reported via a single line of text which can be somewhat cryptic. In addition, we plan to add a single-step facility for ANTLR-generated parsers that can dynamically display the portions of the syntax diagram used in the parse and the parse tree built so far.

CONCLUSIONS

In this paper, we introduce ANTLR, the parser generator of PCCTS. First and foremost, ANTLR is a practical, programmer-friendly tool with many convenient features. ANTLR integrates the specification of lexical and syntactic analysis, supports extended BNF notation, can automatically construct abstract syntax trees, reports and recovers from syntax errors automatically, and provides significant semantic flexibility. ANTLR generates fast, compact, readable recursive-descent parsers in C or C++ which are easy to integrate with other applications.

ANTLR uses a new parsing strategy that makes it possible to develop natural, easy-to-read grammars for difficult languages like C++. ANTLR uses $pred\text{-}LL(k)$ grammars, which are

$LL(k)$ grammars for $k > 1$ augmented with predicates. Predicates allow arbitrary semantic and syntactic information to direct the parse. Due to its power and convenience, ANTLR has over 1000 known users in 37 countries and has become perhaps the second-most popular parser generator both commercially and academically (with YACC/bison being the leader).

ANTLR is free, public-domain software. ANTLR and the rest of PCCTS are available via anonymous `ftp` at `everest.ee.umn.edu` in the directory `pub/pccts` or by sending e-mail to `pccts@ecn.purdue.edu`. In the `pub/pccts/papers` directory, the file `predicates.ps.Z` is compressed postscript for the paper [4]. The newsgroup for ANTLR and PCCTS is `comp.compilers.tools.pccts`. Finally, this paper describes ANTLR as of version 1.30.

ACKNOWLEDGEMENTS

Will Cohen and Hank Dietz were coauthors of the original PCCTS as a whole. Gary Funck at Intrepid Technology, Inc. did extensive testing of ANTLR and provided a constant stream of excellent suggestions. Tom Moog has written a remarkable set of `NOTES.newbie` introductory notes. Ariel Tamches spent a week of his Christmas vacation in the wilds of Minnesota helping with the C++ output. Thom Wood and Randy Helzerman both influenced the C++ output. Anthony Green at Visible Decisions, John Hall at Worcester Polytechnic Institute, Devin Hooker at Ellery Systems, Kenneth D. Weinert at Information Handling Services, Steve Hite, and Roy Levow at Florida Atlantic University have been faithful beta testers of PCCTS. Scott Haney at Lawrence Livermore Labs developed the Macintosh MPW port. Sumana Srinivasan, Mike Monegan, and Steve Naroff of NeXT, Inc. provided extensive help in the definition of the ANTLR C++ output and developed the C++ grammar to be provided with PCCTS. Cathy Tanner proofread several drafts of this paper. Finally, the multitude of PCCTS users have helped refine ANTLR with their suggestions.

REFERENCES

1. S. C. Johnson, *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories; Murray Hill, NJ, 1978.
2. T.J. Parr, H.G. Dietz, and W.E. Cohen, 'PCCTS 1.00: The Purdue Compiler Construction Tool Set', *SIGPLAN Notices*, **27**, (2), 88–165, (February 1992).
3. Terence Parr, Russell Quong, and Hank Dietz, 'The Use of Predicates In $LL(k)$ And $LR(k)$ Parser Generators', Technical Report TREE93-25, Purdue University School of Electrical Engineering, (July 1993).
4. Terence J. Parr and Russell W. Quong, 'Adding Semantic and Syntactic Predicates to $LL(k)$—*pred-*$LL(k)$', *Proceedings of the International Conference on Compiler Construction*, Edinburgh, Scotland, April 1994.
5. Sumana Srinivasan, Steve Naroff, and Mike Monegan. Private communications at NeXT Computer, Incorporated, October 1993.
6. Donald Knuth, 'On the Translation of Languages from Left to Right', *Information and Control*, **8**, 607–639, (1965).
7. Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison Wesley Publishing Company, Reading, Massachusetts, 1990.
8. Terence Parr, Will Cohen, and Hank Dietz, 'The Purdue Compiler Construction Tool Set: Version 1.10 Release Notes', Technical Report Preprint No. 93-088, Army High Performance Computing Research Center, (August 1993).
9. P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns, 'Attributed Translations', *Journal of Computer and System Sciences*, **9**, 279–307, (1974).
10. Charles N. Fischer and Richard J. LeBlanc, *Crafting a Compiler with C*, Benjamin/Cummings Publishing Company, Redwood City, CA, 1991.

APPENDIX: GRAMMATICAL STRUCTURE OF RULES

The following grammar is an ANTLR description of ANTLR's rule meta-language. Much of the complete ANTLR grammar, including the lexical rules and semantic actions, has been removed for clarity, but the full source is available at the ftp site mentioned above.

```
rule:   NONTERMINAL { "!" } { ARGBLK_IN_OR_OUT } { "\>" ARGBLK_IN_OR_OUT }
        { STRING } ":"
        block ";"
        { ACTION }
        ( exception_group )*
    ;

block
    :   alt ( exception_group )* ( "\|" alt ( exception_group )* )*
    ;

alt :   ( { "\~" } element )*
    ;

element_label
    :   ID ":"
    ;

element
    :   { element_label }
        (   TERMINAL { ".." TERMINAL } { "^" | "!" }
        |   "." { "^" | "!" }
        |   NONTERMINAL { "!" } { ARGBLK_IN_OR_OUT } { "\>" ARGBLK_IN_OR_OUT }
        )
    |   ACTION
    |   SEMANTIC_PREDICATE
    |   "\(" block "\)" { "\*" | "\+" | "?" }
    |   "\{" block "\}"
    ;

exception_group
    :   "exception" { LABEL_ID } ( exception_handler )*
        { "default" ":" ACTION }
    ;

exception_handler
    :   "catch" ID ":" { ACTION }
    ;
```

APPENDIX: SAMPLE PROJECTS

To substantiate our claims of broad usage, we asked the users on the pccts mailing list to provide synopses of their projects. Here are edited versions of the first 18 replies in the order they arrived.

Gary Funck, Intrepid Technology Inc. gary@intrepid.com	Pascal to Ada Translator.
Ken Weinert Information Handling Services kenw@ihs.com	SGML translation to vendor data format and User language for specifying data translation from one form to another.
Jim Studt The Federated Software Group, Inc.	Compiler for Forms Interface Management System (a proposed ISO,ANSI standard) target for NCSC B1 mandatory access controlled systems.
David Seidel Innovative Data Concepts Incorporated 71333.1575@compuserve.com	We have used ANTLR/DLG to create the parser for the MAKE engine that we've written for Symantec for inclusion in the next major release of their C++ compiler system.
Kerr Hatrick National Institute for Medical Research k-hatric@nimr.mrc.ac.uk	The production of a protein parser to analyze and categorize protein secondary structure given a protein family grammar.
Tom Zougas Mechanical Engineering, U of Toronto zougas@me.utoronto.ca	I am currently using PCCTS as a command language interpreter as a user interface with an inhouse developed (my PhD) numerical analysis package (nonlinear finite element analysis).
Boleslaw Ciesielski, Viewlogic Sys., Inc. bolek@viewlogic.com	An extension language linked to all of the company's products (CAE applications) and used for extending their functionality and UI.
Peter Dahl University of Minnesota dahl@ee.umn.edu	I use the same ANTLR grammar (to parse DLX assembly) for a code scheduler/Alpha code converter and for a DLX compiled instruction level simulator. I also use ANTLR for my C front end for my compiler.
Sriram Sankar Sun Microsystems Labs, Inc. sriram.sankar@sun.com	The application, ADLT, is a software testing environment based on easy to use formal specifications. ANTLR is used to generate three independent parsers and is used in its wide character mode.
Ivan M Kissiov Cadence Design Systems, Inc. ivan@cadence.com	1. Parser for Analog Hardware Description Language (not yet commercially released). 2. Translator from Analog Behavioral Modeling Language (PROFILE) to Analog Hardware Description Language (not yet released).
Philip A. Wilsey University of Cincinnati phil.wilsey@uc.edu	VHDL parser, code reorganizer, and code generation. Rewriting and backend code generation for semantic modeling project supported by ARPA and Air Force. LL(2) grammar.
Niall Ross Bell Northern Research N.F.Ross@bnr.co.uk	Our application parses SDL (System Description Language: a specification and design language much used in telecoms) output by TeleLOGIC's SDT tool, and rewrites it as GFIF, the language of the SES/workbench performance modeling tool, thus allowing models designed in SDT to be automatically input to SES/workbench for performance analysis.
Steve Robenalt, Scobenalt Engineering robenalt@orange.digex.net	(1) Oberon-2 Compiler for OS/2 under X86 and PowerPC architectures.
Steve Robenalt Rockwell International steve@molly.dny.rockwell.com	1) FORTRAN Translator/Preprocessor (ANTLR, DLG, SORCERER) 2) Plotting Program Command Interpreter (ANTLR, DLG) 3) Graphics Database Translator (ANTLR, DLG)
Vladimir Bacvanski, Aachen Univ of Tech, (Germany) vladimir@ i3.informatik.rwth-aachen.de	A language layer over C++ introducing explicit definition of events and rules for development of multiparadigm systems (.i.e. a fully integrable forward chaining expert system using the C++ data model).
James Mansion Westongold Ltd jgm@cox.compulink.co.uk	I use it for describing interest rate derivative deals and for implementing command line parsers and query and manipulation languages in my dealing support software.
Dana Hoggatt Interactive natural language mathematics muck@mdbs.com	I tell the computer what I want to calculate, and it tells me the answer. No keyboard. No screen. All done via voice recognition and speech synthesis. I'm investigating "verbal" programming techniques, which are radically different from most of the "visual" programming languages used today.
Glen Gordon, Anderson School Of Mgmt. ggordon@AGSM.UCLA.EDU	Translate specially formatted text files into Lotus 123 spreadsheets, formulas and all.