Parsers, Part IV: A Java Cross-Reference Tool

Scott Stanchfield and Terence Parr, MageLang Institute

Have you ever wanted to build a tool that examines or processes Java source files? For example, you might be interested in a tool that enforces a coding style or a tool that automatically augments programs with debugging code snippets. Building a real parser for the Java language with a complete symbol table is nontrivial and most programmers have shied away from such a task and relied on partial solutions using PERL instead.

This is the last article in the parsing series. We built a Java language cross-referencing tool that accumulates complete symbol table information using a Java parser generated by the ANTLR parser generator. The previous article in this series discussed the generation of *tag files* needed to build a code browser and made use of an older version of ANTLR that generated parsers implemented in C++. The tag file generator was almost exclusively a Java parser, with no symbol table manager, and hence, our discussion concentrated on the Java grammar. This article discusses some of the key concepts involved in building a cross-reference tool, focusing on the design of a Java symbol table manager and its interaction with a Java parser. This article also includes a small detour that describes the latest version of ANTLR, generating parsers implemented in the Java language.

A cross-referencing tool assists navigation of code by reporting interactions of modules, variables and classes within that code. It's also useful for determining portions of your code that are no longer used, thereby enabling you to produce the smallest possible program footprint.

If you are unfamiliar with the parsing terms in this article, read Introduction to Parsers, Part III: Does a Yak Have Antlers? by Greg Voss.

What is a Cross-Reference Tool?

A Java program consists of *definitions* and *references*. When you write a program, you define constructs like classes, methods and variables. These definitions include *references* to other Java language constructs. Classes refer to another class as their superclass; variables refer to a class or primitive as their type; some statements refer to methods to delegate processing.

A cross-reference tool describes these relationships by parsing one or more Java source files and determining which constructs are being referred to. These relationships are presented in a report to describe which constructs refer to other constructs and where a given construct is used. An example input and output is provided below.

There are two main components in a cross-reference tool: a parser and a symbol table. The cross-reference parser is similar to the parser described in the previous article in this series. The symbol table is a new feature and is discussed in great detail in this article.

What Functionality Does the Tool Provide?

The Java language cross-reference tool described in this article performs the following functions:

- Support full JDK 1.1 syntax, including inner classes.
- Accept via the command line a list of directories to search. This search is recursive through directories.

- Parse each Java source file in the directories specified definitions and references. Note that Java source files are recognized as any files that have `.java` as a suffix.
- Create a symbol table that contains a list of all definitions in the source files and references to those definitions.
- Generate a report of all defined symbols and where those symbols are referenced.
- Ignore compiled Java bytecode contained in `.class` files.

Because this tool only parses Java source files, some definitions will be unavailable. For example, if a source file references `java.awt.Color`, and you will not be parsing `java/awt/Color.java` during this run of the tool, the tool cannot properly resolve the reference. This is a bit of a problem because some method invocations cannot be properly resolved. For example:

```
java.util.Vector v = new java.util.Vector();
hashTable.put(java.awt.Color.blue);
```

Suppose that `Hashtable` had two different `put` methods. One takes an `Object` and one takes a `Component`. (It doesn't, but suspend disbelief for a moment.) The correct fit cannot be determined because the superclass of `Color` is not known. For that matter the tool cannot determine that `Color.blue` is a `Color`...

Therefore, the tool employs a slightly weaker method resolution algorithm. Assuming the Java source code is valid, chances are pretty good that the number of parameters is sufficient to find the correct method. Obviously this won't always work, but most methods are not overridden and a good number that are differ by number of parameters.

Of course there are many more things that this tool could do; see Possible Extensions for some ideas.

A Sample Input and Output

The following example shows a simple Java class to demonstrate the cross-reference tool's output. When run with the following input:

```
package simple;

import java.awt.Frame;

public class Scrubby extends Frame {
    int y;

    public Scrubby() {
        f(15);
    }

    int f(int x) {
        return x+y;
    }
}
```

the cross-reference tool generates the following report:

```
e:/test/ # java JavaXref simple
Parsing...
   e:\test\simple\Scrubby.java
Resolving types...
resolving: simple.Scrubby
Writing report...

Cross Reference Report
======================
Package: simple
-----------------------------------------
    simple.Scrubby (Class) [Scrubby.java:5]
      Superclass: Frame [Scrubby.java:3]
      Imported classes/packages:
        Frame

      simple.Scrubby.~constructor~ (Method) [Scrubby.java:8]

      simple.Scrubby.y (Variable)  [Scrubby.java:6]
        Type: int
        Referenced:
          File: e:\test\simple\Scrubby.java  Line: 13
      simple.Scrubby.f (Method) [Scrubby.java:12]
        Referenced:
          File: e:\test\simple\Scrubby.java  Line: 9
        Return type: int
        Parameters:
          simple.Scrubby.f.x (Variable)  [Scrubby.java:12]
            Type: int
            Referenced:
              File: e:\test\simple\Scrubby.java  Line: 13
```

A Simple Symbol Table for the Java Language

Java programs are composed of *definitions* and *references*. You define classes, methods and variables and reference them in statements and other definitions. Each class, method and variable has a name. When used in a parser these names are called *symbols*. A symbol represents one specific entity defined in your program. Multiple symbols might appear to be the same, but *are* separate definitions based on their location in the source files. The cross-reference tool needs to keep track of every symbol that is defined and where those symbols are referenced.

A parser tracks symbol information in a data structure called a *symbol table*. The symbol table contains a list of all the Java packages it has encountered, a table of all unique strings and a Stack of scoped-definitions that represent the current parse state. The cross-reference tool uses a class called `SymbolTable` to represent the symbol table.

Every symbol is defined within a certain *scope*. A scope is a section of code over which a definition is visible and usable. Scopes can be nested. For example, a class defines a scope that contains all the variables and methods defined in that class. A method defines a scope that contains all its parameters and local variable definitions. A stack is commonly used to represent this scope nesting. The innermost scope, containing the text being parsed, is always at the top of the stack. As the parse proceeds into new scopes, the new scopes are pushed onto the stack. As the parse exits scopes, the scope definition is popped from the stack. One of the key elements of name lookup is examining each element of the stack, starting at the top, to see if that scope contains the requested name.

All symbol definitions are represented in the symbol table as `Definition` objects. `Definition` is an abstract class that represents some basic, common information about all symbols in the symbol table. Java language constructs such as classes, interfaces and methods all have subclasses that ultimately extend `Definition`. Most of the classes in package `com.magelang.symtab` are extended from `Definition` and provide added information based on the Java construct they represent.

There are two unusual class definitions in this symbol table that are worth mentioning: `PrimitiveDef` and `MultiDef`. `PrimitiveDef` represents a primitive type in the source program. `PrimitiveDef` is a subclass of `ClassDef`. This is not really *correct* from a modeling standpoint. However, subclassing `ClassDef` has a great advantage: widening primitive type conversions can be treated exactly the same as widening class conversions. This simplifies the code for method-parameter type checking because it avoids the special case of primitive-typed parameters, treating all parameters as objects. Note that because only the number of parameters determines method resolution, this advantage is not used. It is presented to show a possible extension to the tool.

The other interesting case is the `MultiDef` class. `MultiDef` collects all definitions in a scope that have the same name, such as overloaded methods or a method that has the same name as a class variable. `MultiDef` defines an interesting method lookup algorithm that selects the appropriate method from a set of applicable methods for a given method call.

During the parse the symbol table is loaded with information about symbol definitions taken from the context of those definitions. This information is used later to find which symbol a name in the source code refers to and store that reference information for later inclusion in the cross-reference report.

The class diagram and descriptions of all classes will help you understand how the symbol table classes are related. The symbol table described might seem complex, but it is very simple compared to many, especially for C++.

How to Collect the Symbol Information

There are several possible designs for the overall processing necessary to build the symbol table and resolve references. The Java language allows classes to be defined *after* they have been used in a source file. To properly resolve a reference to a class, a tool can use one of the following strategies:

- *Walk, then resolve*. During the parse, watch for definitions and references. Definitions are recorded in the symbol table and references are recorded as strings. After the parse has been completed for all files, all definitions have been recorded in the symbol table. Then, walk through all symbols in the symbol table and resolve the references.
- *Backpatch unknown definitions*. This technique uses placeholder objects in the symbol table to track symbols that have not yet been defined. When the unknown symbol is finally defined, the parser will replace the placeholder references with the reference to the real definition.
  The difference between this and the first approach is that there is not a full walk through the symbol table to resolve references; references are resolved as definitions are encountered.

Backpatching is the more efficient approach, but adds a great deal of complexity to the process. For simplicity, the cross-reference tool described in this article implements the first strategy.

The tool performs its processing in three phases: parse the source code to determine definitions and references; resolve references in the contents of the symbol table; and generate the cross-reference report. Here are the phases described in more detail:

1. **Determine Definitions and Note References**
   The first phase of the cross-reference tool walks through all the source code in the specified directories and their subdirectories. The parser collects information on the following constructs:
   - Package Definitions
   - Class Definitions
   - Interface Definitions
   - Method Definitions
   - References to other symbols

For each of the above definitions found, a new symbol is created. Some symbols, like classes, reference other symbols such as a superclass that is being extended as part of a class definition. During the first pass, these references might not yet be available; they could be defined later in the same source file or in another file that has yet to be parsed. Because of this, any references to superclasses, implemented interfaces, return types for methods and parameter types are stored as placeholders by their names only. These placeholders are held in an instance of the `DummyClass` class. These will also be resolved during phase two.

2. **Resolve Definition References**
   The result of the source-code parse is a symbol table that lists all constructs defined in the source files. Some of these definitions reference other definitions, for superclasses, variable types and so on. This second phase will walk through all definitions in the symbol table and resolve those references. Most of the symbol table classes implement a method called `resolveTypes` that is used to perform this resolution.
   At the conclusion of this pass, the symbol table contains all symbols defined in the parsed source files and proper resolutions to defined symbols.

3. **Report Generation**
   This final phase looks at the data contained in the symbol table and generates a cross-reference report.

Writing the Code

Consider the thought process that goes into writing a tool like this. A tutorial is beyond the scope of this article, but here are a few key concepts and methods for the tool's implementation.

The tool was developed with the following steps:

1. Select the tools.
2. Write a recognizer.
3. Design and implement a symbol table.
4. Add actions for definitions to the grammar.
5. Add actions for references to the grammar.
6. Resolve references
7. Create report generation routines.

Selecting the Tools

The first choice was to decide which tools to use to create the cross-reference tool. Tools, in this instance, means the language and parser generator. The Java language was a natural choice and this narrowed the scope of available parser generation tools. Fortunately ANTLR 2.0 exists and is an excellent choice for that parser generator!

Writing a Recognizer

The tool began as a simple JDK 1.1 recognizer. The LALR(1) grammar for the Java language, as defined in *The Java Language Specification* (JLS), was converted to a valid LL(*k*) grammar, suitable for use in ANTLR 2.0. For information on how to perform this conversion, see Converting Grammars from LALR to LL.

The recognizer grammar defines the basic scanner and parser. This allows recognition of any valid Java compilation unit. This recognizer is similar to the browser described in the last article, without any embedded action code. For more information on recognition, see Parsers, Part III: A Parser for the Java Language.

There are two scripts that assist compilation and execution of the recognizer. Note that you will need to have installed ANTLR 2.0 per its installation instructions. Two versions of the scripts are provided, for Unix and DOS users. In Unix, issue the following commands from a command shell:

```
build
runnit name1 name2 name3 ... nameN
```

name1 through *nameN* are the names of files or directories to be recognized. In DOS, use `build.bat` and `runnit.bat` in the same manner.

Note that this grammar file constitutes a separate tool from the cross-reference tool, one that checks syntax of Java source files.

Designing and Implementing a Symbol Table

A recognizer is nice, but it has no memory of what was parsed. That's where a symbol table is helpful. The tool needs to track and resolve references. The addition of a symbol table allows definition tracking, which in turn provides enough information for symbol resolution.

Symbol table design is *not* trivial. There are several common patterns used, but symbol tables for each language tend to differ greatly. The following sections describe some of the requirements for the Java symbol table and some basic algorithms to implement those requirements.

How to Store Data

Java packages provide a good anchor for all other data in the symbol table. A Java package can contain classes and interfaces. Classes and interfaces can contain methods, variables, inner classes and interfaces. These are but a few of the relationships between symbol objects. See Symbol Table Classes for a more complete description.

The symbol table stores a `Vector` of all parsed package definitions. This `Vector` allows access to any symbol defined in the parsed source files. The symbol table also contains a stack of scopes that represent the current parse state.

Looking up Symbol Definitions

One of the basic constructs in nearly every symbol table is a stack of scopes. Each scope that you enter while parsing a source file gets pushed onto this stack, giving you a simple lookup mechanism for *normal* variables. Suppose you were writing a symbol table for a language like Pascal that has nested procedures and no inheritance. The lookup problem is simple: when the parser sees *x*, is that *x* in the current scope or in some scope containing the current scope? This simple lookup algorithm looks like this:

```
lookup1(name):
  for each scope on the scope stack (starting at the top)
      if name is defined in that scope
          it's been found!
      otherwise, try the next enclosing scope
```

This is the basis of symbol lookup in nearly every computer language. Note that the cross-reference tool doesn't define a `lookup1` method; this action is built into the `SymbolTable`'s lookup method. Method `lookup1` is used here as an example to make the processing easier to see.

Looking up Qualified Names

To complicate matters a bit, many languages allow explicit qualification of names. For example, a name like `x.y` in a given language might mean that `y` must be looked up in the scope of `x`. The lookup algorithm was changed slightly by allowing a specific scope *and only* that specific scope to be searched. When the parser recognizes `x`, the *next* symbol is looked up in `x`'s scope. This requires a routine named `setScope` that sets a variable in the symbol table to the specific scope to search. When the parser sees `x.y`, the following processing occurs:

```
  setScope(lookup("x"));
  lookup("y");
```

and `lookup` is defined as

```
  lookup(name):
  if a specific scope is set
      look in that scope for name
      clear the specific scope
      return the result
  else
      lookup1(name)
```

Adding Inheritance

Inheritance provides another complication to symbol lookup. This really isn't as bad as it seems, though. The trick is to modify the `lookup1` method:

```
lookup1(name):
  for each scope s on the scope stack (starting at the top)
      scope to search = s
      while not found and s is valid
          if name is defined in s
              it's been found!
          else if s has a superclass
              s = superclass of s
          else
              s = NOT VALID
      otherwise, try the next containing scope
```

Inheritance lookup is achieved by nesting an inheritance search *inside* the normal scoped search. A similar modification is made to the `lookup` method and similar processing is used to lookup a name in an implemented interface. Note that the interface lookup proceeds *after* the inheritance lookup.

Looking up Overloaded Symbols

What to do about overloading? In the Java language, several methods can have the same name or the same name as a variable in the class. Each method is unique based on the types of the parameters to that method. A proper method lookup modifies the above algorithms to achieve the following effect:

```
  get the set of all methods that are applicable
  bestSofar = first applicable method
  for each other applicable method challenger
      if challenger is more specific than bestSoFar
          bestSoFar = challenger
```

An *applicable* method is one where the *actual parameter* types are either exact matches for the method's *formal parameter* types or can be widened to the formal parameters. Widening refers to a *widening conversion*; see the *Java Language Specification* (*JLS*) for details. In short, think actual parameters must be subclasses or same type as formal parameters. Consult the *JLS* for details on Primitive type conversions.

A method is *more specific* if at least one of its formal parameter types can be widened to a parameter type in the other method. A simple, but sneaky, way to test this is to see if one method is applicable for the *other method's* formal parameter types. If so, the other method is more specific.

Because of limited access to all superclasses of all types the tool needed an alternate method of resolving overloaded methods. Instead of using the above algorithm, it was simplified to a simple match of the number of parameters. This is effective for most methods, but fails in cases where overloaded methods have the same number of parameters.

Storage in the symbol table consists primarily of hash tables that store other elements. Each symbol that defines a scope contains a hash table of all elements in its scope. This works well if all elements have unique names. However, elements that have the same name, such as overloaded methods, require an alternate storage mechanism.

That's where the `MultiDef` class comes into play. `MultiDef` is a list of other `Definitions`. If the name lookup returns a `MultiDef`, that `MultiDef` resolves the symbol further using the number of parameters, or -1 if looking for a variable reference.

Now that you've had a high-level overview of the symbol table processing, take a look at the symbol table source code.

Adding Actions for Definitions to the Grammar

Having covered the background information, it is time to discuss the interaction between the recognizer and the symbol table. The tool contains an instance of the symbol table in the parser and calls to store all definition and reference information in that symbol table. If you look at the source code for the final cross reference grammar, you'll see the symbol table being created in the `main` method and passed to each parser that is created. A separate *parser* instance is created for each file to be parsed, but they all share the same *symbol table*. The parser also defines several convenience methods that just pass data to the symbol table. This approach allows easy addition of debugging code in these wrapper methods to track the data that is being passed to the symbol table.

The first thing necessary for tracking definition information is to expand the common token type that is used by the recognizer. The recognizer uses `antlr.CommonToken` as the object that is passed from the scanner to the parser representing a token. The symbol table needs a little information passed to it from the parser. Extending `CommonToken`, creating a new class named `JavaToken`, adds a `File` reference and parameter count, information that can be used when resolving referenced symbols.

Next, annotate the grammar with action code that stores Java language constructs in the symbol table. Here are a few examples of language construct definitions, but this is by no means an exhaustive list:

- `classDefinition`
- `packageDefinition`
- `variableDeclarator`
- `compoundStatement`
- `methodHead`

At each place where a Java language construct is recognized, add code to inform the symbol table. Using `methodHead` as an example:

```
// This is the header of a method.  It includes the name and parameters
//   for the method.
//   This also watches for a list of exception classes in a throws clause.
methodHead
  :  IDENT  // the name of the method

     // parse the formal parameter declarations.
     LPAREN (parameterDeclarationList)? RPAREN

     // again, the array specification is skipped...
     (LBRACK RBRACK)*

     // get the list of exceptions that this method is declared to throw
    (throwsClause)?
  ;
```

After seeing the method name, give the symbol table enough information to define a `MethodDef` object. The symbol table also needs to associate the parameters with the method.

To create the method definition, pass the following information to the symbol table:

- Method name
- Return type

But the return type definition is in another rule! In the LL(*k*) grammar required the `type` be factored out because `field` could be a method or variable definition, as well as a few other things:

```
field :
  ...
  typeSpec  // method or variable declaration(s)
  (  methodHead ( compoundStatement | SEMI )
  |  variableDefinitions SEMI
  )
```

`typeSpec` must reside in field, so how can it be available in `methodHead`? There are two possibilities:

- Pass the type information *into* `methodHead`.
- Return the other method information *from* `methodHead` and define the method in `field`.

The first alternative is preferable, as it helps keep the method definition call with the method match. So, modify `typeSpec` to return a `JavaToken` and pass that token up through `type` to `typeSpec`.

```
typeSpec returns [JavaToken t]
  {t=null;}
  :  t=type (LBRACK RBRACK )*
  ;
```

Then modify `field` to use that type:

```
field
  {JavaToken type;}
  :  // method, constructor or variable declaration

  ...

  | type=typeSpec  // method or variable declaration(s)
    ( methodHead[type]
       ( compoundStatement[BODY] | SEMI {popScope();})
       |  variableDefinitions[type] SEMI
```

```
        )
```

Rule `field` grabs the type and passes it down into `methodHead`. Meanwhile, `methodHead` needed to be changed to accept the type and pass the data to the symbol table:

```
  methodHead[JavaToken type]
    {JavaVector exceptions=null;} // to keep track of thrown exceptions
    :  method:IDENT  // the name of the method

       // tell the symbol table about it.  Note that this signals that
       // we are in a method header so we handle parameters appropriately
       {defineMethod((JavaToken)method, type);}

       // parse the formal parameter declarations.  These are sent to the
       // symbol table as _variables_.  Because the symbol table knows we
       // are in a method header, it collects these definitions as parameters
       // for the method.
       LPAREN (parameterDeclarationList)? RPAREN

       // again, the array specification is skipped...
       (LBRACK RBRACK)*

       // get the list of exceptions that this method is declared to throw
       (exceptions=throwsClause)?

       // tell the symbol table we are done with the method header. Note that
       // this will tell the symbol table to handle variables normally
       {endMethodHead(exceptions);}
    ;
```

Method `defineMethod` tells the symbol table about the newly found method definition. It also sets a symbol table flag indicating that any variable definitions will be added as parameters to the current method header. Method `endMethodHeader` halts that behavior. Similar actions need to be added anywhere the grammar recognizes that a construct is defined in a Java program.

As a check, define a few dump methods in the symbol table. Run the parser and dump the symbol table contents to ensure that all definitions were added properly. This is an important thing to do at this point, because you must make sure your definitions are properly added to the symbol table before trying to add and resolve references.

Adding Actions for References to the Grammar

Now the symbol table needs actions that tell it about symbol references. Be careful here, as it's very easy to have references double up if you have a reference call in a low-level rule and in one that eventually calls that low-level rule. The places where you need to flag references are

- labels for `break` and `continue` statements.
- the type in a cast expression.
- the type in an `instanceof` expression.
- the id in a `postfixExpression`.

Other references are set up automatically by the symbol table. For example, when defining a class that has a superclass, a reference to the superclass is automatically set up.

Reference actions are very simple. Take, for example, the following code that is defined in rule `statement`.

```
  // get out of a loop (or switch)
  // tell the symbol table that we are (possibly) referencing a label
  |  "break" (bid:IDENT {reference((JavaToken)bid);})? SEMI
```

The `reference` method tells the symbol table about the token, which contains location information, and the symbol table simply stores the reference in the current scope's unresolved reference list.

Resolving References

The symbol table also needed to check all of its definitions and see which have unresolved references. These references could be a superclass for a class or a variable used in an expression.

Some symbols have several things to resolve, which, in turn reference other symbols. It can be very difficult things to properly resolving references without entering an infinite loop, especially with circular definitions. In some cases a variable tracks when an object, such as a `Vector`, is currently being resolved, preventing circular infinite resolution attempts.

The symbol table classes define methods named `resolveTypes` and in some cases `resolveRefs`. Examine these methods to see what was done to perform this resolution. In particular, examine the `lookup` methods defined in `SymbolTable` and how they deal with qualified names.

An important note: the language specification is ***extremely*** important, especially with regard to name lookup. Whenever you are writing a parser, *do not* assume you know the language you are parsing. Thoroughly read the language specification to ensure you see all the details *before* starting to write your grammar. See *The Java Language Specification* for the exact definition of the Java language.

Creating Report Generation Routines

Writing a readable report is the final task. Many people prefer nicely indented reports, where indentation shows scope containment of other definitions. To facilitate this, define a class named `IndentingPrintWriter` that subclasses `PrintWriter`. Override all the `print` and `println` methods to write a certain padding string before anything on a line and defined an `indent` and `dedent` method to change the amount of padding.

Finally, define a `report(IndentingPrintWriter)` method in each symbol table class to print information about the class, indent the print writer, report information about contained elements and dedent the print writer.

Where is the Source Code?

All of the source code for the cross-reference tool is provided. There are two scripts that assist compilation and execution of the cross-reference tool. Note that you will need to have installed ANTLR 2.0 per its installation instructions. Two versions of the scripts are provided, for Unix and DOS users. In Unix, issue the following commands from a command shell:

```
build
runnit name1 name2 name3 ... nameN
```

name1 through *nameN* are the names of files or directories to be recognized. In DOS, use `build.bat` and `runnit.bat` in the same manner.

Symbol Table Classes describes each of the classes used in the symbol table and has links to each symbol table file independently.

All source code presented in this article is free for your use with no restrictions whatsoever. However, there is no guarantee of the code's suitability for any purpose. Use of this code is at your own risk.

ANTLR 2.00, or *PCCTS: Part Deux*

The Java parser in the previous article of this parsing series was written using *PCCTS* (The Purdue Compiler Construction Tool Set), a tool with a long history going back to the late 1980's. PCCTS could only generate C or C++. *ANTLR* (ANother Tool for Language Recognition) is the next generation of PCCTS, and is a complete rewrite of the tool.   ANTLR is written in and generates parsers in Java source code. The most significant change between the two tools involves a radical departure from conventional lexical analysis.

ANTLR generalizes the notion of scanning, parsing and tree parsing into the single abstraction of applying grammatical structure to an input stream, be it a stream of characters, tokens or tree nodes. A grammar file in ANTLR now consists of grammar class definitions using a single consistent notation rather than different notations for parsing and scanning. Language tools have traditionally used different notations for the seemingly different tasks. For example, programmers used *regular expression* notation to describe input symbols and BNF (Backus-Naur Form) notation to describe language structure. In other words, programmers had to use the regular expression

```
[a-z]+
```

meaning one or more characters from ‘a’ to ‘z’, to describe an identifier and

```
declaration
  : type IDENTIFIER ASSIGN expression SEMICOLON
  ;
```

to describe the grammatical structure of a declaration in their language of interest. ANTLR uses the same extended-BNF notation to describe both. So, in a lexical grammar, you can specify:

```
class MyPascalLexer extends Lexer;
  …
  IDENTIFIER
    : ( 'a'..'z' )+
    ;
```

Along with this wonderful unification, ANTLR generates much stronger scanners than the finite-automata scanners generated by traditional tools such as `lex` and DLG, the lexer-generator in PCCTS. The ANTLR scanner mechanism is better because you can do the following:

- Have semantic and syntactic predicates and use k>1 lookahead.
- Read and debug the output as it's very similar to what you would build by hand; that is, it's not a table full of integers like automata-based scanners.
- Have actions executed during the recognition of a single token.
- Recognize complicated tokens such as HTML tags or special comments like the `javadoc` @-tags inside `/** ... */` comments. The lexer has a stack, unlike a finite automaton, so you can match nested structures such as nested comments.

The increase in scanning power comes at the cost of some inconvenience in scanner specification and indeed requires a shift in your thoughts about lexical analysis. Finite automata do automatic left-factoring of common, possibly infinite, left-prefixes. In a hand-built scanner, you have to find and factor out all common prefixes. For example, consider writing a lexer to match integers and floats. The regular expressions are straightforward:

```
INTEGER:"[0-9]+" ;
```

```
REAL:"[0-9]+{.[0-9]*}|.[0-9]+" ;
```

Building a scanner for this by hand or using ANTLR's scanner mechanism requires factoring out the common `[0-9]+`. In practice, however, that the advantages seriously outweigh this one inconvenience.

For a full description of ANTLR's lexical mechanism, see the documentation and resources at the ANTLR Home Page.

Conclusion

The cross-reference tool described in this article is a *real* tool and demonstrates the power of parsing technology. Cross-referencing is a common task and provides a glimpse into several issues related to parser development. A real tool is not just a simple grammar that recognizes input of a given source language; it needs other constructs that work in concert with the parser. The latest version of ANTLR provides an ideal implementation of the tool, parsing Java source files using the Java language itself. Symbol tables are an incredibly powerful resource, adding memory to a parser and resolving references between definitions in a parsed file.

Additional Information

ANTLR - ANother Tool for Language Recognition
    http://java.magelang.com/antlr
The Java Home Page
    http://java.sun.com
The Java Language Specification
    http://java.sun.com/docs/books/jls/html
Converting Grammars from LALR to LL
    http://www.scruz.net/~thetick/lalrtoll.html
ANTLR Usenet Newsgroup
    news:comp.compilers.tools.pccts
General Compilers/Parsers Newsgroup
    news:comp.compilers