[CUP Logo Image]

CUP User's Manual

Scott E. Hudson
Graphics Visualization and Usability Center
Georgia Institute of Technology

Modified by Frank Flannery, C. Scott Ananian, Dan Wang with advice from Andrew W. Appel
Now actualized by Michael Petter
Last updated March 2006 (v0.11a)

Table of Contents

i. About CUP Version 0.10
1. Introduction and Example
2. Specification Syntax
3.1 Running CUP
3.2 CUP and ANT
4. Customizing the Parser
5. Scanner interface
5.1 Basic Symbol management
5.2 Advanced Symbol management
6. Error Recovery
7. Conclusion
References
A. Grammar for CUP Specification Files
B. A Very Simple Example Scanner
C. Incompatibilites between CUP 0.9 and CUP 0.10
D.

Bugs
E.
Change log

i. About CUP Version 0.10

Version 0.10 of CUP adds many new changes and features over the previous releases of version 0.9. These changes attempt to make CUP more like its predecessor, YACC. As a result, the old 0.9 parser specifications for CUP are not compatible and a reading of appendix C of the new manual will be necessary to write new specifications. The new version, however, gives the user more power and options, making parser specifications easier to write.

ii. About CUP Version 0.11

in version 0.11 the TUM team tries to continue the success story of CUP 0.10, beginning with the introduction of generic data types for non-terminal symbols as well as a modernisation of the user interface with a comfortable ANT plugin structure.

1. Introduction and Example

This manual describes the basic operation and use of the Java(tm) Based Constructor of Useful Parsers (CUP for short). CUP is a system for generating LALR parsers from simple specifications. It serves the same role as the widely used program YACC [1] and in fact offers most of the features of YACC. However, CUP is written in Java, uses specifications including embedded Java code, and produces parsers which are implemented in Java.

Although this manual covers all aspects of the CUP system, it is relatively brief, and assumes you have at least a little bit of knowledge of LR parsing. A working knowledge of YACC is also very helpful in understanding how CUP specifications work. A number of compiler construction textbooks (such as [2,3]) cover this material, and discuss the YACC system (which is quite similar to this one) as a specific example.

Using CUP involves creating a simple specification based on the grammar for which a parser is needed, along with construction of a scanner capable of breaking characters up into meaningful tokens (such as keywords, numbers, and special symbols).

As a simple example, consider a system for evaluating simple arithmetic expressions over integers. This system would read expressions from standard input (each terminated with a semicolon), evaluate them, and print the result on standard output. A grammar for the input to such a system might look like:

```
expr_list ::= expr_list expr_part | expr_part
expr_part ::= expr ';'
expr      ::= expr '+' expr | expr '-' expr | expr '*' expr
            | expr '/' expr | expr '%' expr | '(' expr ')'
            | '-' expr | number
```

To specify a parser based on this grammar, our first step is to identify and name the set of terminal symbols that will appear on input, and the set of non-terminal symbols. In this case, the non-terminals are:

```
expr_list, expr_part  and  expr .
```

For terminal names we might choose:

```
  SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD, NUMBER, LPAREN,
and RPAREN
```

The experienced user will note a problem with the above grammar. It is ambiguous. An ambiguous grammar is a grammar which, given a certain input, can reduce the parts of the input in two different ways such as to give two different answers. Take the above grammar, for example. given the following input:
`3 + 4 * 6`
The grammar can either evaluate the `3 + 4` and then multiply seven by six, or it can evaluate `4 * 6` and then add three. Older versions of CUP forced the user to write unambiguous grammars, but now there is a construct allowing the user to specify precedences and associativities for terminals. This means that the above ambiguous grammar can be used, after specifying precedences and associativities. There is more explanation later. Based on these namings we can construct a small CUP specification as follows:

```
// CUP specification for a simple expression evaluator (no actions)

import java_cup.runtime.*;

/* Preliminaries to set up and use the scanner.  */
init with {: scanner.init();              :};
scan with {: return scanner.next_token(); :};

/* Terminals (tokens returned by the scanner). */
terminal            SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal            UMINUS, LPAREN, RPAREN;
terminal Integer    NUMBER;

/* Non terminals */
non terminal            expr_list, expr_part;
non terminal Integer    expr, term, factor;

/* Precedences */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;

/* The grammar */
expr_list ::= expr_list expr_part |
              expr_part;
expr_part ::= expr SEMI;
expr      ::= expr PLUS expr
            | expr MINUS expr
            | expr TIMES expr
            | expr DIVIDE expr
            | expr MOD expr
            | MINUS expr %prec UMINUS
            | LPAREN expr RPAREN
            | NUMBER
            ;
```

We will consider each part of the specification syntax in detail later. However, here we can quickly see that the specification contains four main parts. The first part provides preliminary and miscellaneous declarations to specify how the parser is to be generated, and supply parts of the runtime code. In this case we indicate that the `java_cup.runtime` classes should be imported, then supply a small bit of initialization code, and some code for invoking the scanner to retrieve the next input token. The second part of the specification declares terminals and non-terminals, and associates object classes with each. In this case, the terminals are declared as either with no type, or of type `Integer`. The specified type of the terminal or non-terminal is the type of the value of those terminals or non-terminals. If no type is specified, the terminal or non-terminal carries no value. Here, no type indicates that these terminals and non-terminals hold no value. The third part specifies the precedence and associativity of terminals. The last precedence declaration give its terminals the highest precedence. The final part of the specification contains the grammar.

To produce a parser from this specification we use the CUP generator. If this specification were stored in a file `parser.cup`, then (on a Unix system at least) we might invoke CUP using a command like:

```
java -jar java-cup-11a.jar parser.cup
```

In this case, the system will produce two Java source files containing parts of the generated parser: `sym.java` and `parser.java`. As you might expect, these two files contain declarations for the classes `sym` and `parser`. The `sym` class contains a series of constant declarations, one for each terminal symbol. This is typically used by the scanner to refer to symbols (e.g. with code such as "`return new Symbol(sym.SEMI);`"). The `parser` class implements the parser itself.

The specification above, while constructing a full parser, does not perform any semantic actions &emdash; it will only indicate success or failure of a parse. To calculate and print values of each expression, we must embed Java code within the parser to carry out actions at various points. In CUP, actions are contained in *code strings* which are surrounded by delimiters of the form `{:` and `:}` (we can see examples of this in the `init with` and `scan with` clauses above). In general, the system records all characters within the delimiters, but does not try to check that it contains valid Java code.

A more complete CUP specification for our example system (with actions embedded at various points in the grammar) is shown below:

```
// CUP specification for a simple expression evaluator (w/ actions)

import java_cup.runtime.*;

/* Preliminaries to set up and use the scanner.  */
init with {: scanner.init();              :};
scan with {: return scanner.next_token(); :};

/* Terminals (tokens returned by the scanner). */
terminal           SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal           UMINUS, LPAREN, RPAREN;
terminal Integer   NUMBER;

/* Non-terminals */
non terminal            expr_list, expr_part;
non terminal Integer    expr;
```

```
/* Precedences */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;

/* The grammar */
expr_list ::= expr_list expr_part
              |
              expr_part;

expr_part ::= expr:e
              {: System.out.println("= " + e); :}
              SEMI
              ;

expr      ::= expr:e1 PLUS expr:e2
              {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
              |
              expr:e1 MINUS expr:e2
              {: RESULT = new Integer(e1.intValue() - e2.intValue()); :}
              |
              expr:e1 TIMES expr:e2
              {: RESULT = new Integer(e1.intValue() * e2.intValue()); :}
              |
              expr:e1 DIVIDE expr:e2
              {: RESULT = new Integer(e1.intValue() / e2.intValue()); :}
              |
              expr:e1 MOD expr:e2
              {: RESULT = new Integer(e1.intValue() % e2.intValue()); :}
              |
              NUMBER:n
              {: RESULT = n; :}
              |
              MINUS expr:e
              {: RESULT = new Integer(0 - e.intValue()); :}
              %prec UMINUS
              |
              LPAREN expr:e RPAREN
              {: RESULT = e; :}
              ;
```

Here we can see several changes. Most importantly, code to be executed at various points in the parse is included inside code strings delimited by `{:` and `:}`. In addition, labels have been placed on various symbols in the right hand side of productions. For example in:

```
  expr:e1 PLUS expr:e2
        {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
```

the first non-terminal `expr` has been labeled with `e1`, and the second with `e2`. The left hand side value of each production is always implicitly labeled as `RESULT`.

Each symbol appearing in a production is represented at runtime by an object of type `Symbol` on the parse stack. The labels refer to the instance variable `value` in those objects. In the expression `expr:e1 PLUS expr:e2`, `e1` and `e2` refer to objects of type Integer. These objects are in the value fields of the objects of type `Symbol` representing those non-terminals on the parse stack. `RESULT` is of type `Integer` as well, since the resulting non-terminal `expr` was declared as of type `Integer`. This object becomes the `value` instance variable of a new `Symbol` object.

For each label, two more variables accessible to the user are declared. A left and right value labels are passed to the code string, so that the user can find out where the left and right side of each terminal or non-terminal is in the input stream. The name of these variables is the label name, plus `left` or `right`. for example, given the right hand side of a production `expr:e1 PLUS expr:e2` the user could not only access variables `e1` and `e2`, but also `e1left, e1right, e2left` and `e2right`. these variables are of type `int`.

The final step in creating a working parser is to create a *scanner* (also known as a *lexical analyzer* or simply a *lexer*). This routine is responsible for reading individual characters, removing things things like white space and comments, recognizing which terminal symbols from the grammar each group of characters represents, then returning Symbol objects representing these symbols to the parser. The terminals will be retrieved with a call to the scanner function. In the example, the parser will call `scanner.next_token()`. The scanner should return objects of type `java_cup.runtime.Symbol`. This type is very different than older versions of CUP's `java_cup.runtime.symbol`. These Symbol objects contains the instance variable `value` of type Object, which should be set by the lexer. This variable refers to the value of that symbol, and the type of object in value should be of the same type as declared in the `terminal` and `non terminal` declarations. In the above example, if the lexer wished to pass a NUMBER token, it should create a `Symbol` with the `value` instance variable filled with an object of type `Integer`. `Symbol` objects corresponding to terminals and non-terminals with no value have a null value field.

The code contained in the `init with` clause of the specification will be executed before any tokens are requested. Each token will be requested using whatever code is found in the `scan with` clause. Beyond this, the exact form the scanner takes is up to you; however note that each call to the scanner function should return a new instance of `java_cup.runtime.Symbol` (or a subclass). These symbol objects are annotated with parser information and pushed onto a stack; reusing objects will result in the parser annotations being scrambled. As of CUP 0.10j, `Symbol` reuse should be detected if it occurs; the parser will throw an `Error` telling you to fix your scanner.

In the next section a more detailed and formal explanation of all parts of a CUP specification will be given. Section 3 describes options for running the CUP system. Section 4 discusses the details of how to customize a CUP parser, while section 5 discusses the scanner interface added in CUP 0.10j. Section 6 considers error recovery. Finally, Section 7 provides a conclusion.

2. Specification Syntax

Now that we have seen a small example, we present a complete description of all parts of a CUP specification. A specification has four sections with a total of eight specific parts (however, most of these are optional). A specification consists of:

- package and import specifications,
- user code components,
- symbol (terminal and non-terminal) lists,
- precedence declarations, and
- the grammar.

Each of these parts must appear in the order presented here. (A complete grammar for the specification language is given in Appendix A.) The particulars of each part of the specification are described in the subsections below.

Package and Import Specifications

A specification begins with optional `package` and `import` declarations. These have the same syntax, and play the same role, as the package and import declarations found in a normal Java program. A package declaration is of the form:

```
package name;
```

where name *name* is a Java package identifier, possibly in several parts separated by ".". In general, CUP employs Java lexical conventions. So for example, both styles of Java comments are supported, and identifiers are constructed beginning with a letter, dollar sign ($), or underscore (_), which can then be followed by zero or more letters, numbers, dollar signs, and underscores.

After an optional `package` declaration, there can be zero or more `import` declarations. As in a Java program these have the form:

```
import package_name.class_name;
```

or

```
import package_name.*;
```

The package declaration indicates what package the `sym` and `parser` classes that are generated by the system will be in. Any import declarations that appear in the specification will also appear in the source file for the `parser` class allowing various names from that package to be used directly in user supplied action code.

User Code Components

Following the optional `package` and `import` declarations are a series of optional declarations that allow user code to be included as part of the generated parser (see Section 4 for a full description of how the parser uses this code). As a part of the parser file, a separate non-public class to contain all embedded user actions is produced. The first `action code` declaration section allows code to be included in this class. Routines and variables for use by the code embedded in the grammar would normally be placed in this section (a typical example might be symbol table manipulation routines). This declaration takes the form:

```
action code {: ... :};
```

where `{: ... :}` is a code string whose contents will be placed directly within the `action class` class declaration.

After the `action code` declaration is an optional `parser code` declaration. This declaration allows methods and variable to be placed directly within the generated parser class. Although this is less common, it can be helpful when customizing the parser &emdash; it is possible for example, to include scanning methods inside the parser and/or override the default error reporting routines. This declaration is very similar to the `action code` declaration and takes the form:

```
parser code {: ... :};
```

Again, code from the code string is placed directly into the generated parser class definition.

Next in the specification is the optional `init` declaration which has the form:

```
init with {: ... :};
```

This declaration provides code that will be executed by the parser before it asks for the first token. Typically, this is used to initialize the scanner as well as various tables and other data structures that might be needed by semantic actions. In this case, the code given in the code string forms the body of a `void` method inside the `parser` class.

The final (optional) user code section of the specification indicates how the parser should ask for the next token from the scanner. This has the form:

```
scan with {: ... :};
```

As with the `init` clause, the contents of the code string forms the body of a method in the generated parser. However, in this case the method returns an object of type `java_cup.runtime.Symbol`. Consequently the code found in the `scan with` clause should return such a value. See section 5 for information on the default behavior if the `scan with` section is omitted.

As of CUP 0.10j the action code, parser code, init code, and scan with sections may appear in any order. They must, however, precede the symbol lists.

Symbol Lists

Following user supplied code comes the first required part of the specification: the symbol lists. These declarations are responsible for naming and supplying a type for each terminal and non-terminal symbol that appears in the grammar. As indicated above, each terminal and non-terminal symbol is represented at runtime with a `Symbol` object. In the case of terminals, these are returned by the scanner and placed on the parse stack. The lexer should put the value of the terminal in the `value` instance variable. In the case of non-terminals these replace a series of `Symbol` objects on the parse stack whenever the right hand side of some production is recognized. In order to tell the parser which object types should be used for which symbol, `terminal` and `non terminal` declarations are used. These take the forms:

```
terminal classname name1, name2, ...;
non terminal classname name1, name2, ...;
terminal name1, name2, ...;
```

and

```
non terminal name1, name2, ...;
```

where *`classname`* can be a multiple part name separated with "."s. The *`classname`* specified represents the type of the value of that terminal or non-terminal. When accessing these values through labels, the users uses the type declared. the *`classname`* can be of any type. If no *`classname`* is given, then the terminal or non-terminal holds no value. a label referring to such a symbol with have a null value. As of CUP 0.10j, you may specify non-terminals the declaration "`nonterminal`" (note, no space) as well as the original "`non terminal`" spelling.

Names of terminals and non-terminals cannot be CUP reserved words; these include "code", "action", "parser", "terminal", "non", "nonterminal", "init", "scan", "with", "start", "precedence", "left", "right", "nonassoc", "import", and "package".

Precedence and Associativity declarations

The third section, which is optional, specifies the precedences and associativity of terminals. This is useful for parsing with ambiguous grammars, as done in the example above. There are three type of precedence/associativity declarations:

```
precedence left     terminal[, terminal...];
precedence right    terminal[, terminal...];
precedence nonassoc terminal[, terminal...];
```

The comma separated list indicates that those terminals should have the associativity specified at that precedence level and the precedence of that declaration. The order of precedence, from highest to lowest, is bottom to top. Hence, this declares that multiplication and division have higher precedence than addition and subtraction:

```
precedence left  ADD, SUBTRACT;
precedence left  TIMES, DIVIDE;
```

Precedence resolves shift reduce problems. For example, given the input to the above example parser `3 + 4 * 8`, the parser doesn't know whether to reduce `3 + 4` or shift the '*' onto the stack. However, since '*' has a higher precedence than '+', it will be shifted and the multiplication will be performed before the addition.

CUP assigns each one of its terminals a precedence according to these declarations. Any terminals not in this declaration have lowest precedence. CUP also assigns each of its productions a precedence. That precedence is equal to the precedence of the last terminal in that production. If the production has no terminals, then it has lowest precedence. For example, `expr ::= expr TIMES expr` would have the same precedence as `TIMES`. When there is a shift/reduce conflict, the parser determines whether the terminal to be shifted has a higher precedence, or if the production to reduce by does. If the terminal has higher precedence, it it shifted, if the production has higher precedence, a reduce is performed. If they have equal precedence, associativity of the terminal determine what happens.

An associativity is assigned to each terminal used in the precedence/associativity declarations. The three associativities are `left, right` and `nonassoc` Associativities are also used to resolve shift/reduce conflicts, but only in the case of equal precedences. If the associativity of the terminal that can be shifted is `left`, then a reduce is performed. This means, if the input is a string of additions, like `3 + 4 + 5 + 6 + 7`, the parser will *always* reduce them from left to right, in this case, starting with `3 + 4`. If the associativity of the terminal is `right`, it is shifted onto the stack. hence, the reductions will take place from right to left. So, if PLUS were declared with associativity of `right`, the `6 + 7` would be reduced first in the above string. If a terminal is declared as `nonassoc`, then two consecutive occurrences of equal precedence non-associative terminals generates an error. This is useful for comparison operations. For example, if the input string is `6 == 7 == 8 == 9`, the parser should generate an error. If '==' is declared as `nonassoc` then an error will be generated.

All terminals not used in the precedence/associativity declarations are treated as lowest precedence. If a shift/reduce error results, involving two such terminals, it cannot be resolved, as the above conflicts are, so it will be reported.

The Grammar

The final section of a CUP declaration provides the grammar. This section optionally starts with a declaration of the form:

```
start with non-terminal;
```

This indicates which non-terminal is the *start* or *goal* non-terminal for parsing. If a start non-terminal is not explicitly declared, then the non-terminal on the left hand side of the first production will be used. At the end of a successful parse, CUP returns an object of type `java_cup.runtime.Symbol`. This `Symbol`'s value instance variable contains the final reduction result.

The grammar itself follows the optional `start` declaration. Each production in the grammar has a left hand side non-terminal followed by the symbol "`::=`", which is then followed by a series of zero or more actions, terminal, or non-terminal symbols, followed by an optional contextual precedence assignment, and terminated with a semicolon (;).

Each symbol on the right hand side can optionally be labeled with a name. Label names appear after the symbol name separated by a colon (:). Label names must be unique within the production, and can be used within action code to refer to the value of the symbol. Along with the label, two more variables are created, which are the label plus `left` and the label plus `right`. These are `int` values that contain the right and left locations of what the terminal or non-terminal covers in the input file. These values must be properly initialized in the terminals by the lexer. The left and right values then propagate to non-terminals to which productions reduce.

If there are several productions for the same non-terminal they may be declared together. In this case the productions start with the non-terminal and "`::=`". This is followed by multiple right hand sides each separated by a bar (|). The full set of productions is then terminated by a semicolon.

Actions appear in the right hand side as code strings (e.g., Java code inside `{:` ... `:}` delimiters). These are executed by the parser at the point when the portion of the production to the left of the action has been recognized. (Note that the scanner will have returned the token one past the point of the action since the parser needs this extra *lookahead* token for recognition.)

Contextual precedence assignments follow all the symbols and actions of the right hand side of the production whose precedence it is assigning. Contextual precedence assignment allows a production to be assigned a precedence not based on the last terminal in it. A good example is shown in the above sample parser specification:

```
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;

expr ::=  MINUS expr:e
          {: RESULT = new Integer(0 - e.intValue()); :}
          %prec UMINUS
```

Here, there production is declared as having the precedence of UMINUS. Hence, the parser can give the MINUS sign two different precedences, depending on whether it is a unary minus or a subtraction operation.

3. Running CUP

3.1 Command line interface

As mentioned above, CUP is written in Java. To invoke it, one needs to use the Java interpreter to invoke the static method `java_cup.Main()`, passing an array of strings containing options. Assuming a Unix machine, the simplest way to do this is typically to invoke it directly from the command line with a command such as:

```
java -jar java-cup-11a.jar options  inputfile
```

Once running, CUP expects to find a specification file on standard input and produces two Java source files as output.

In addition to the specification file, CUP's behavior can also be changed by passing various options to it. Legal options are documented in `Main.java` and include:

`-package` *name*
: Specify that the `parser` and `sym` classes are to be placed in the named package. By default, no package specification is put in the generated code (hence the classes default to the special "unnamed" package).

`-parser` *name*
: Output parser and action code into a file (and class) with the given name instead of the default of "`parser`".

`-symbols` *name*
: Output the symbol constant code into a class with the given name instead of the default of "`sym`".

`-interface`
: Outputs the symbol constant code as an `interface` rather than as a `class`.

`-nonterms`
: Place constants for non-terminals into the symbol constant class. The parser does not need these symbol constants, so they are not normally output. However, it can be very helpful to refer to these constants when debugging a generated parser.

`-expect` *number*
: During parser construction the system may detect that an ambiguous situation would occur at runtime. This is called a *conflict*. In general, the parser may be unable to decide whether to *shift* (read another symbol) or *reduce* (replace the recognized right hand side of a production with its left hand side). This is called a *shift/reduce conflict*. Similarly, the parser may not be able to decide between reduction with two different productions. This is called a *reduce/reduce conflict*. Normally, if one or more of these conflicts occur, parser generation is aborted. However, in certain carefully considered cases it may be advantageous to arbitrarily break such a conflict. In this case CUP uses YACC convention and resolves shift/reduce conflicts by shifting, and reduce/reduce conflicts using the "highest priority" production (the one declared first in the specification). In order to enable automatic breaking of conflicts the `-expect` option must be given indicating exactly how many conflicts are expected. Conflicts resolved by precedences and associativities are not reported.

`-compact_red`
: Including this option enables a table compaction optimization involving reductions. In particular, it allows the most common reduce entry in each row of the parse action table to be used as the default for that row. This typically saves considerable room in the tables, which can grow to be very large. This optimization has the effect of replacing all error entries in a row with the default reduce entry. While this may sound dangerous, if not down right incorrect, it turns out that this does not affect the correctness of the parser. In particular, some changes of this type are inherent in LALR parsers (when compared to canonical LR parsers), and the resulting parsers will still never read past the first token at which the error could be detected. The parser can, however, make extra erroneous reduces before detecting the error, so this can degrade the parser's ability to do error recovery. (Refer to reference [2] pp. 244-247 or reference [3] pp. 190-194 for a complete explanation of this compaction technique.)

  This option is typically used to work-around the java bytecode limitations on table initialization code sizes. However, CUP 0.10h introduced a string-encoding for the parser tables which is not subject to the standard method-size limitations. Consequently, use of this option should no longer be required for large grammars.

`-nowarn`
: This options causes all warning messages (as opposed to error messages) produced by the system to be suppressed.

`-nosummary`
: Normally, the system prints a summary listing such things as the number of terminals, non-terminals, parse states, etc. at the end of its run. This option suppresses that summary.

`-progress`
: This option causes the system to print short messages indicating its progress through various parts of the parser

generation process.

`-dump_grammar`
`-dump_states`
`-dump_tables`
`-dump`
: These options cause the system to produce a human readable dump of the grammar, the constructed parse states (often needed to resolve parse conflicts), and the parse tables (rarely needed), respectively. The `-dump` option can be used to produce all of these dumps.

`-time`
: This option adds detailed timing statistics to the normal summary of results. This is normally of great interest only to maintainers of the system itself.

`-debug`
: This option produces voluminous internal debugging information about the system as it runs. This is normally of interest only to maintainers of the system itself.

`-nopositions`
: This option keeps CUP from generating code to propagate the left and right hand values of terminals to non-terminals, and then from non-terminals to other terminals. If the left and right values aren't going to be used by the parser, then it will save some runtime computation to not generate these position propagations. This option also keeps the left and right label variables from being generated, so any reference to these will cause an error.

`-noscanner`
: CUP 0.10j introduced improved scanner integration and a new interface, `java_cup.runtime.Scanner`. By default, the generated parser refers to this interface, which means you cannot use these parsers with CUP runtimes older than 0.10j. If your parser does not use the new scanner integration features, then you may specify the `-noscanner` option to suppress the `java_cup.runtime.Scanner` references and allow compatibility with old runtimes. Not many people should have reason to do this.

`-version`
: Invoking CUP with the `-version` flag will cause it to print out the working version of CUP and halt. This allows automated CUP version checking for Makefiles, install scripts and other applications which may require it.

3.2 Integrating CUP into an

ANT script

To use cup in an ANT script, You have to add the following task definition to Your `build.xml` file:

```
<taskdef name="cup"
  classname="java_cup.anttask.CUPTask"
  classpathref="cupclasspath"
/>
```

Now, You are ready to use Your new <cup/> task to generate own parsers from within ANT. Such a generation statement could look like:

```
<target name="cup">
<cup srcfile="path/to/cupfile/Parser.cup"
  destdir="path/to/javafiles"
  interface="true"
/>
</target>
```

You can specify all commandline flags from chapter 3.1 as boolean parameters to Your cuptask to achieve a similar behaviour (as done with `-interface` in this little example).

4. Customizing the Parser

Each generated parser consists of three generated classes. The `sym` class (which can be renamed using the `-symbols` option) simply contains a series of `int` constants, one for each terminal. Non-terminals are also included if the `-nonterms` option is given. The source file for the `parser` class (which can be renamed using the `-parser` option) actually contains two class definitions, the public `parser` class that implements the actual parser, and another non-public class (called `CUP$action`) which encapsulates all user actions contained in the grammar, as well as code from the `action code` declaration. In addition to user supplied code, this class contains one method: `CUP$do_action` which consists of a large switch statement for selecting and executing various fragments of user supplied action code. In general, all names beginning with the prefix of `CUP$` are reserved for internal uses by CUP generated code.

The `parser` class contains the actual generated parser. It is a subclass of `java_cup.runtime.lr_parser` which implements a general table driven framework for an LR parser. The generated `parser` class provides a series of tables for use by the general framework. Three tables are provided:

the production table
: provides the symbol number of the left hand side non-terminal, along with the length of the right hand side, for each production in the grammar,

the action table
: indicates what action (shift, reduce, or error) is to be taken on each lookahead symbol when encountered in each state, and

the reduce-goto table
: indicates which state to shift to after reduces (under each non-terminal from each state).

(Note that the action and reduce-goto tables are not stored as simple arrays, but use a compacted "list" structure to save a significant amount of space. See comments the runtime system source code for details.)

Beyond the parse tables, generated (or inherited) code provides a series of methods that can be used to customize the generated parser. Some of these methods are supplied by code found in part of the specification and can be customized directly in that fashion. The others are provided by the `lr_parser` base class and can be overridden with new versions (via the `parser code` declaration) to customize the system. Methods available for customization include:

`public void user_init()`
: This method is called by the parser prior to asking for the first token from the scanner. The body of this method contains the code from the `init with` clause of the the specification.

`public java_cup.runtime.Symbol scan()`
: This method encapsulates the scanner and is called each time a new terminal is needed by the parser. The body of this method is supplied by the `scan with` clause of the specification, if present; otherwise it returns `getScanner().next_token()`.

`public java_cup.runtime.Scanner getScanner()`
: Returns the default scanner. See section 5.

`public void setScanner(java_cup.runtime.Scanner s)`
: Sets the default scanner. See section 5.

`public void report_error(String message, Object info)`
: This method should be called whenever an error message is to be issued. In the default implementation of this method, the first parameter provides the text of a message which is printed on `System.err` and the second parameter is simply ignored. It is very typical to override this method in order to provide a more sophisticated

error reporting mechanism.

`public void report_fatal_error(String message, Object info)`

This method should be called whenever a non-recoverable error occurs. It responds by calling `report_error()`, then aborts parsing by calling the parser method `done_parsing()`, and finally throws an exception. (In general `done_parsing()` should be called at any point that parsing needs to be terminated early).

`public void syntax_error(Symbol cur_token)`

This method is called by the parser as soon as a syntax error is detected (but before error recovery is attempted). In the default implementation it calls: `report_error("Syntax error", null);`.

`public void unrecovered_syntax_error(Symbol cur_token)`

This method is called by the parser if it is unable to recover from a syntax error. In the default implementation it calls: `report_fatal_error("Couldn't repair and continue parse", null);`.

`protected int error_sync_size()`

This method is called by the parser to determine how many tokens it must successfully parse in order to consider an error recovery successful. The default implementation returns 3. Values below 2 are not recommended. See the section on error recovery for details.

Parsing itself is performed by the method `public Symbol parse()`. This method starts by getting references to each of the parse tables, then initializes a `CUP$action` object (by calling `protected void init_actions()`). Next it calls `user_init()`, then fetches the first lookahead token with a call to `scan()`. Finally, it begins parsing. Parsing continues until `done_parsing()` is called (this is done automatically, for example, when the parser accepts). It then returns a `Symbol` with the `value` instance variable containing the RESULT of the start production, or `null`, if there is no value.

In addition to the normal parser, the runtime system also provides a debugging version of the parser. This operates in exactly the same way as the normal parser, but prints debugging messages (by calling `public void debug_message(String mess)` whose default implementation prints a message to `System.err`).

Based on these routines, invocation of a CUP parser is typically done with code such as:

```
/* create a parsing object */
parser parser_obj = new parser();

/* open input files, etc. here */
Symbol parse_tree = null;

try {
  if (do_debug_parse)
    parse_tree = parser_obj.debug_parse();
  else
    parse_tree = parser_obj.parse();
} catch (Exception e) {
  /* do cleanup here - - possibly rethrow e */
} finally {
  /* do close out here */
}
```

5. Scanner Interface

5.1 Basic Symbol management

In CUP 0.10j, scanner integration was improved according to suggestions made by David MacMahon. The changes make it easier to incorporate JLex and other automatically-generated scanners into CUP parsers.

To use the new code, your scanner should implement the `java_cup.runtime.Scanner` interface, defined as:

```
package java_cup.runtime;

public interface Scanner {
    public Symbol next_token() throws java.lang.Exception;
}
```

In addition to the methods described in section 4, the `java_cup.runtime.lr_parser` class has two new accessor methods, `setScanner()` and `getScanner()`. The default implementation of `scan()` is:

```
  public Symbol scan() throws java.lang.Exception {
    return getScanner().next_token();
  }
```

The generated parser also contains a constructor which takes a `Scanner` and calls `setScanner()` with it. In most cases, then, the `init with` and `scan with` directives may be omitted. You can simply create the parser with a reference to the desired scanner:

```
      /* create a parsing object */
      parser parser_obj = new parser(new my_scanner());
```

or set the scanner after the parser is created:

```
      /* create a parsing object */
      parser parser_obj = new parser();
      /* set the default scanner */
      parser_obj.setScanner(new my_scanner());
```

Note that because the parser uses look-ahead, resetting the scanner in the middle of a parse is not recommended. If you attempt to use the default implementation of `scan()` without first calling `setScanner()`, a `NullPointerException` will be thrown.

As an example of scanner integration, the following three lines in the lexer-generator input are all that is required to use a JLex scanner with CUP:

```
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
```

It is anticipated that the JLex directive `%cup` will abbreviate the above three directive in the next version of JLex. Invoking the parser with the JLex scanner is then simply:

```
parser parser_obj = new parser( new Yylex( some_InputStream_or_Reader));
```

Note that you still have to handle EOF correctly; the JLex code to do so is something like:

```
%eofval{
  return sym.EOF;
%eofval}
```

where `sym` is the name of the symbol class for your generated parser.

The simple_calc example in the CUP distribution illustrates the use of the scanner integration features with a hand-coded scanner.

5.2 Advanced Symbol management

Since CUP v11a we offer the possibility of advanced symbol handling in CUP. Therefore, You can implement Your own SymbolFactory, derived from `java_cup.runtime.SymbolFactory`, and have CUP manage Your own type of symbols. We've done that for You already in the pre-defined `java_cup.runtime.ComplexSymbolFactory`, which provides support for detailed location information in the symbol class. Just have a look at CUPs own `Lexer.jflex`, which is already using the new feature.

All You have to do is providing Your CUP-generated parser with the new SymbolFactory which can be done like this:

```
SymbolFactory symbolFactory = new ComplexSymbolFactory();
MyParser parser = new MyParser(new Lexer(inputfile,symbolFactory),
symbolFactory);
```

Also, You can use the factory methods in Your SymbolFactory to have callbacks/hooks into the semantic action methods. That is especially usefull, when You want to equip Your syntax tree with Location information as You can do as follows:

```
public Symbol newSymbol(String name, Symbol left, Symbol right, Object value){
   ComplexSymbol sym = (ComplexSymbol)super.newSymbol(name,left,right,value);
   SyntaxTreeNode node = (SyntaxTreeNode) value;
   node.setLeft(left.getLeft());
   node.setRight(right.getRight());
   return sym;
}
```

6. Error Recovery

A final important aspect of building parsers with CUP is support for syntactic error recovery. CUP uses the same error recovery mechanisms as YACC. In particular, it supports a special error symbol (denoted simply as `error`). This symbol plays the role of a special non-terminal which, instead of being defined by productions, instead matches an erroneous input sequence.

The error symbol only comes into play if a syntax error is detected. If a syntax error is detected then the parser tries to replace some portion of the input token stream with `error` and then continue parsing. For example, we might have productions such as:

```
    stmt ::= expr SEMI | while_stmt SEMI | if_stmt SEMI | ... |
             error SEMI
             ;
```

This indicates that if none of the normal productions for `stmt` can be matched by the input, then a syntax error should be declared, and recovery should be made by skipping erroneous tokens (equivalent to matching and replacing them with `error`) up to a point at which the parse can be continued with a semicolon (and additional context that legally follows a statement). An error is considered to be recovered from if and only if a sufficient number of tokens past the `error` symbol can be successfully parsed. (The number of tokens required is determined by the `error_sync_size()` method of the parser and defaults to 3).

Specifically, the parser first looks for the closest state to the top of the parse stack that has an outgoing transition under `error`. This generally corresponds to working from productions that represent more detailed constructs (such as a specific kind of statement) up to productions that represent more general or enclosing constructs (such as the general production for all statements or a production representing a whole section of declarations) until we get to a place where an error recovery production has been provided for. Once the parser is placed into a configuration that has an immediate error recovery (by popping the stack to the first such state), the parser begins skipping tokens to find a point at which the parse can be continued. After discarding each token, the parser attempts to parse ahead in the input (without executing any embedded semantic actions). If the parser can successfully parse past the required number of tokens, then the input is backed up to the point of recovery and the parse is resumed normally (executing all actions). If the parse cannot be continued far enough, then another token is discarded and the parser again tries to parse ahead. If the end of input is reached without making a successful recovery (or there was no suitable error recovery state found on the parse stack to begin with) then error recovery fails.

7. Conclusion

This manual has briefly described the CUP LALR parser generation system. CUP is designed to fill the same role as the well known YACC parser generator system, but is written in and operates entirely with Java code rather than C or C++. Additional details on the operation of the system can be found in the parser generator and runtime source code. See the CUP home page below for access to the API documentation for the system and its runtime.

This document covers version 0.10j of the system. Check the CUP home page: http://www.cs.princeton.edu/~appel/modern/java/CUP/ for the latest release information, instructions for downloading the system, and additional news about CUP. Bug reports and other comments for the developers should be sent to the CUP maintainer, C. Scott Ananian, at cananian@alumni.princeton.edu

CUP was originally written by Scott Hudson, in August of 1995.

It was extended to support precedence by Frank Flannery, in July of 1996.

On-going improvements have been done by C. Scott Ananian, the CUP maintainer, from December of 1997 to the present.

References

[1]
S. C. Johnson, "YACC &emdash; Yet Another Compiler Compiler", CS Technical Report #32, Bell Telephone Laboratories, Murray Hill, NJ, 1975.

[2]
A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing, Reading, MA, 1986.

[3]
C. Fischer, and R. LeBlanc, *Crafting a Compiler with C*, Benjamin/Cummings Publishing, Redwood City, CA,

1991.

Appendix A. Grammar for CUP Specification Files (0.10j)

```
java_cup_spec      ::= package_spec import_list code_parts
                       symbol_list precedence_list start_spec
                       production_list
package_spec       ::= PACKAGE multipart_id SEMI | empty
import_list        ::= import_list import_spec | empty
import_spec        ::= IMPORT import_id SEMI
code_part          ::= action_code_part | parser_code_part |
                       init_code | scan_code
code_parts         ::= code_parts code_part | empty
action_code_part   ::= ACTION CODE CODE_STRING opt_semi
parser_code_part   ::= PARSER CODE CODE_STRING opt_semi
init_code          ::= INIT WITH CODE_STRING opt_semi
scan_code          ::= SCAN WITH CODE_STRING opt_semi
symbol_list        ::= symbol_list symbol | symbol
symbol             ::= TERMINAL type_id declares_term |
                       NON TERMINAL type_id declares_non_term |
                       NONTERMINAL type_id declares_non_term |
                       TERMINAL declares_term |
                       NON TERMINAL declares_non_term |
                       NONTERMIANL declared_non_term
term_name_list     ::= term_name_list COMMA new_term_id | new_term_id
non_term_name_list ::= non_term_name_list COMMA new_non_term_id |
                       new_non_term_id
declares_term      ::= term_name_list SEMI
declares_non_term  ::= non_term_name_list SEMI
precedence_list    ::= precedence_l | empty
precedence_l       ::= precedence_l preced + preced;
preced             ::= PRECEDENCE LEFT terminal_list SEMI
                       | PRECEDENCE RIGHT terminal_list SEMI
                       | PRECEDENCE NONASSOC terminal_list SEMI
terminal_list      ::= terminal_list COMMA terminal_id | terminal_id
start_spec         ::= START WITH nt_id SEMI | empty
production_list    ::= production_list production | production
production         ::= nt_id COLON_COLON_EQUALS rhs_list SEMI
rhs_list           ::= rhs_list BAR rhs | rhs
rhs                ::= prod_part_list PERCENT_PREC term_id |
                       prod_part_list
prod_part_list     ::= prod_part_list prod_part | empty
prod_part          ::= symbol_id opt_label | CODE_STRING
opt_label          ::= COLON label_id | empty
multipart_id       ::= multipart_id DOT ID | ID
import_id          ::= multipart_id DOT STAR | multipart_id
type_id            ::= multipart_id
terminal_id        ::= term_id
term_id            ::= symbol_id
```

```
new_term_id        ::= ID
new_non_term_id    ::= ID
nt_id              ::= ID
symbol_id          ::= ID
label_id           ::= ID
opt_semi           ::= SEMI | empty
```

Appendix B. A Very Simple Example Scanner

```
// Simple Example Scanner Class

import java_cup.runtime.*;
import sym;

public class scanner {
  /* single lookahead character */
  protected static int next_char;
  // since cup v11 we use SymbolFactories rather than Symbols
  private SymbolFactory sf = new DefaultSymbolFactory();

  /* advance input by one character */
  protected static void advance()
    throws java.io.IOException
    { next_char = System.in.read(); }

  /* initialize the scanner */
  public static void init()
    throws java.io.IOException
    { advance(); }

  /* recognize and return the next complete token */
  public static Symbol next_token()
    throws java.io.IOException
    {
      for (;;)
        switch (next_char)
          {
            case '0': case '1': case '2': case '3': case '4':
            case '5': case '6': case '7': case '8': case '9':
              /* parse a decimal integer */
              int i_val = 0;
              do {
                i_val = i_val * 10 + (next_char - '0');
                advance();
              } while (next_char >= '0' && next_char <= '9');
            return sf.newSymbol("NUMBER",sym.NUMBER, new Integer(i_val));
```

```
            case ';': advance(); return sf.newSymbol("SEMI",sym.SEMI);
            case '+': advance(); return sf.newSymbol("PLUS",sym.PLUS);
            case '-': advance(); return sf.newSymbol("MINUS",sym.MINUS);
            case '*': advance(); return sf.newSymbol("TIMES",sym.TIMES);
            case '/': advance(); return sf.newSymbol("DIVIDE",sym.DIVIDE);
            case '%': advance(); return sf.newSymbol("MOD",sym.MOD);
            case '(': advance(); return sf.newSymbol("LPAREN",sym.LPAREN);
            case ')': advance(); return sf.newSymbol("RPAREN",sym.RPAREN);

            case -1: return return sf.newSymbol("EOF",sym.EOF);

            default:
              /* in this simple scanner we just ignore everything else */
              advance();
            break;
          }
    }
};
```

Appendix C: Incompatibilites between CUP 0.9 and CUP 0.10

CUP version 0.10a is a major overhaul of CUP. The changes are severe, meaning no backwards compatibility to older versions. The changes consist of:

- A different lexical interface,
- New terminal/non-terminal declarations,
- Different label references,
- A different way of passing RESULT,
- New position values and propagation,
- Parser now returns a value,
- Terminal precedence declarations and
- Rule contextual precedence assignment

Lexical Interface

CUP now interfaces with the lexer in a completely different manner. In the previous releases, a new class was used for every distinct type of terminal. This release, however, uses only one class: The `Symbol` class. The `Symbol` class has three instance variables which are significant to the parser when passing information from the lexer. The first is the `value` instance variable. This variable contains the value of that terminal. It is of the type declared as the terminal type in the parser specification file. The second two are the instance variables `left` and `right`. They should be filled with the `int` value of where in the input file, character-wise, that terminal was found.

For more information, refer to the manual on scanners.

Terminal/Non-Terminal Declarations

Terminal and non-terminal declarations now can be declared in two different ways to indicate the values of the terminals or non-terminals. The previous declarations of the form

```
terminal classname terminal [, terminal ...];
```

still works. The classname, however indicates the type of the value of the terminal or non-terminal, and does not indicate the type of object placed on the parse stack. A declaration, such as:

```
terminal terminal [, terminal ...];
```

indicates the terminals in the list hold no value.

For more information, refer to the manual on declarations.

Label References

Label references do not refer to the object on the parse stack, as in the old CUP, but rather to the value of the `value` instance variable of the `Symbol` that represents that terminal or non-terminal. Hence, references to terminal and non-terminal values is direct, as opposed to the old CUP, where the labels referred to objects containing the value of the terminal or non-terminal.

For more information, refer to the manual on labels.

RESULT Value

The `RESULT` variable refers directly to the value of the non-terminal to which a rule reduces, rather than to the object on the parse stack. Hence, `RESULT` is of the same type the non-terminal to which it reduces, as declared in the non-terminal declaration. Again, the reference is direct, rather than to something that will contain the data.

For more information, refer to the manual on RESULT.

Position Propagation

For every label, two more variables are declared, which are the label plus `left` or the label plus `right`. These correspond to the left and right locations in the input stream to which that terminal or non-terminal came from. These values are propagated from the input terminals, so that the starting non-terminal should have a left value of 0 and a right value of the location of the last character read.

For more information, refer to the manual on positions.

Return Value

A call to `parse()` or `debug_parse()` returns a Symbol. This Symbol is the start non-terminal, so the `value` instance variable contains the final `RESULT` assignment.

Precedence

CUP now has precedenced terminals. a new declaration section, occurring between the terminal and non-terminal declarations and the grammar specifies the precedence and associativity of rules. The declarations are of the form:

```
precedence {left| right | nonassoc} terminal[, terminal ...];
...
```

The terminals are assigned a precedence, where terminals on the same line have equal precedences, and the precedence declarations farther down the list of precedence declarations have higher precedence. `left, right` and `nonassoc` specify the associativity of these terminals. left associativity corresponds to a reduce on conflict, right to a shift on conflict, and nonassoc to an error on conflict. Hence, ambiguous grammars may now be used.

For more information, refer to the manual on precedence.

Contextual Precedence

Finally the new CUP adds contextual precedence. A production may be declare as followed:

```
lhs ::= {right hand side list of terminals, non-terminals and actions}
        %prec {terminal};
```

this production would then have a precedence equal to the terminal specified after the `%prec`. Hence, shift/reduce conflicts can be contextually resolved. Note that the `%prec` *terminal* part comes after all actions strings. It does not come before the last action string.

For more information, refer to the manual on contextual precedence. These changes implemented by:

Frank Flannery
Department of Computer Science
Princeton University

Appendix D: Bugs

In this version of CUP it's difficult for the semantic action phrases (Java code attached to productions) to access the `report_error` method and other similar methods and objects defined in the `parser code` directive.

This is because the parsing tables (and parsing engine) are in one object (belonging to class `parser` or whatever name is specified by the **-parser** directive), and the semantic actions are in another object (of class `CUP$actions`).

However, there is a way to do it, though it's a bit inelegant. The action object has a `private final` field named `parser` that points to the parsing object. Thus, methods and instance variables of the parser can be accessed within semantic actions as:

```
parser.report_error(message,info);
x = parser.mydata;
```

Perhaps this will not be necessary in a future release, and that such methods and variables as `report_error` and `mydata` will be available directly from the semantic actions; we will achieve this by combining the "parser" object and the "actions" object together.

For a list of any other currently known bugs in CUP, see http://www.cs.princeton.edu/~appel/modern/java/CUP/bugs.html.

Appendix E: Change log

0.9e
: March 1996, Scott Hudson's original version.

0.10a
: August 1996, several major changes to the interface.

0.10b
: November 1996, fixes a few minor bugs.

0.10c
: July 1997, fixes a bug related to precedence declarations.

0.10e
: September 1997, fixes a bug introduced in 0.10c relating to `nonassoc` precedence. Thanks to Tony Hosking for reporting the bug and providing the fix. Also recognizes carriage-return character as white space and fixes a number of other small bugs.

0.10f
: December 1997, was a maintenance release. The CUP source was cleaned up for JDK 1.1.

0.10g
: March 1998, adds new features and fixes old bugs. The behavior of RESULT assignments was normalized, and a problem with implicit start productions was fixed. The CUP grammar was extended to allow array types for terminals and non-terminals, and a command-line flag was added to allow the generation of a symbol *interface*, rather than class. Bugs associated with multiple invocations of a single parser object and multiple CUP classes in one package have been stomped on. Documentation was updated, as well.

0.10h-0.10i
: February 1999, are maintenance releases.

0.10j
: July 1999, broadened the CUP input grammar to allow more flexibility and improved scanner integration via the `java_cup.runtime.Scanner` interface.

0.11a
: the changelog has moved to the internet to sustain a more up-to-date state.
