Copper version 0.7
User Manual

August Schwerdfeger
(schwerdf@cs.umn.edu)
Minnesota Extensible Language Tools
University of Minnesota

Contents

Chapter 1

Introduction.

1.1 Copper in a nutshell.

This manual contains instructions on how to use Copper, a Java-based LALR(1) parser generator with expanded parsing capability compared to the Java-based CUP (`http://www2.cs.tum.edu/projects/cup/`) or the C-based Bison (`http://www.gnu.org/software/bison/`) .

Like CUP and Bison, Copper takes the specification of a formal grammar and generates from it a program (specifically, a Java class) that can parse the language of that grammar. However, unlike CUP and Bison, Copper gives you everything necessary to do so. Parsers from most generators require an external scanner built by another tool, a scanner generator — JLex (`http://www.cs.princeton.edu/~appel/modern/java/JLex/`) is usually used with CUP and Flex (`http://flex.sourceforge.net/`) with Bison — in order to work. Copper generates both the scanner and the parser from a single specification and puts them in a single Java class; this integration enables Copper to parse a larger class of grammars than CUP or Bison.

This manual assumes a working knowledge of LR parsing; knowledge of CUP and JLex may also be helpful.

1.2 Example specification.

Copper is designed to support several "skins," or methods of input, to suit a wide range of grammar writers. There are presently three such skins. The first is a skin mimicking the input styles of JLex and CUP as closely as possible, meant for use by flesh-and-blood grammar writers.

The other two, an XML schema and an API, are meant for use with machine-generated grammar specifications. They are not covered in this manual, but are instead documented in Copper's Javadoc (`http://melt.cs.umn.edu/copper/0.7/javadoc/`).

Input to Copper consists, loosely, of preamble materials (package and import declarations, *etc.*), lexical syntax (terminal symbols and regexes used to build the scanner) and context-free syntax (nonterminal symbols and productions used to build the parser). Semantic actions may optionally be supplied with productions and terminals.

For an example of a grammar specification written for the CUP skin of Copper, see Algorithms 1 (no semantic actions) and 2 (with semantic actions). The parser compiled for this grammar will recognize arithmetic operations over integers, by the four arithmetic operations as well as unary negation. Note that there are features of this specification that would not be found in grammar specifications written for traditional tools, such as two terminals sharing the same regex; these will be discussed in further detail below.

1.2.1 Overview of manual.

The structure of the rest of the manual is as follows. Chapter 2 discusses the novel features of Copper and how to utilize them. Chapter 3 contains further information about the CUP skin, including the general organization of a specification in the CUP skin and the syntax of each construct it supports. Chapter 4 contains information about running Copper, such as command-line syntax and how to interpret errors. Chapter 5 contains information about utilizing the generated parser. Appendix A contains the grammar of the CUP skin's concrete syntax, while Appendix B contains a more elaborate example of Copper's use in the form of a grammar for the "Mini-Java" language from Andrew Appel's *Modern Compiler Implementation in Java*. Appendix C gives information about Copper's software licensing.

Algorithm 1 Recognizer for simple arithmetic grammar.

```
package math;
/* This is a RECOGNIZER for a simple arithmetic
 * language; it will give errors when invalid
 * strings are entered, but takes no action on valid
 * strings. */
%%
%parser ArithmeticParser
/* Lexical syntax */
%lex{

    /* Whitespace */
    ignore terminal WS ::= /[ ]+/;
    /* Grammar terminals */
    terminal PLUS           ::= /\+/;
    terminal UNARY_MINUS    ::= /-/;
    terminal BINARY_MINUS   ::= /-/;
    terminal TIMES          ::= /\*/;
    terminal DIVIDE         ::= /\//;
    terminal LPAREN         ::= /\(/;
    terminal RPAREN         ::= /\)/;
    terminal NUMBER         ::= /0|([1-9][0-9]*)/;

%lex}
/* Context-free syntax */
%cf{

    /* Nonterminals */
    non terminal expr;
    /* Start symbol */
    start with expr;
    /* Precedences */
    precedence left PLUS, BINARY_MINUS;
    precedence left TIMES, DIVIDE;
    precedence left UNARY_MINUS;
    expr ::=
          expr PLUS expr
        | expr BINARY_MINUS expr
        | expr TIMES expr
        | expr DIVIDE expr
        | UNARY_MINUS expr
           %layout ()
        | LPAREN expr RPAREN
        | NUMBER
        ;

%cf}
```

Algorithm 2 Parser for simple arithmetic grammar.

```
package math;
/* This is a PARSER for a simple arithmetic
 * language; when run on a valid string, it
 * will return the value of the expression
 * represented by that string. */
%%
%parser ArithmeticParser
/* Lexical syntax */
%lex{

    /* Whitespace */
    ignore terminal WS ::= /[ ]+/;
    /* Grammar terminals */
    terminal PLUS           ::= /\+/;
    terminal UNARY_MINUS    ::= /-/;
    terminal BINARY_MINUS   ::= /-/;
    terminal TIMES          ::= /\*/;
    terminal DIVIDE         ::= /\//;
    terminal LPAREN         ::= /\(/;
    terminal RPAREN         ::= /\)/;
    terminal Integer NUMBER ::= /0|([1-9][0-9]*)/
    {:
        RESULT = Integer.parseInt(lexeme);
    :};

%lex}
/* Context-free syntax */
%cf{

    /* Nonterminals */
    non terminal Integer expr;
    /* Start symbol */
    start with expr;
    /* Precedences */
    precedence left PLUS, BINARY_MINUS;
    precedence left TIMES, DIVIDE;
    precedence left UNARY_MINUS;
    expr ::=

        expr:l PLUS expr:r          {: RESULT = l + r; :}
        | expr:l BINARY_MINUS expr:r {: RESULT = l - r; :}
        | expr:l TIMES expr:r       {: RESULT = l * r; :}
        | expr:l DIVIDE expr:r      {: RESULT = l / r; :}
        | UNARY_MINUS expr:e        {: RESULT = -1 * e; :}
           %layout ()
        | LPAREN expr:e RPAREN      {: RESULT = e; :}
        | NUMBER:n                  {: RESULT = n; :}
        ;

%cf}
```

Chapter 2

Copper's parsing algorithms.

2.1 Context-aware scanning.

The most crucial difference between Copper and the standard LALR(1) parser generator is the addition of *context-aware scanning*.

The typical scanner will scan through the input file and separate it into a stream of tokens with no feedback from the parser. A scanner in Copper, by contrast, contains a distinct sub-scanner for every state of the parser; scanner and parser work in lock-step, and for each token a different scanner will run, scanning only for those terminals that are valid syntax at that location.[1]

This enables such constructs as the two "minus" terminals in the arithmetic grammar of Algorithm 1; `UNARY_MINUS` occurs *before* an expression, while `BINARY_MINUS` occurs *between* expressions. However, it also requires more careful planning of lexical syntax, as described in Sections 2.2 and 2.3.

2.2 Specification of whitespace and other layout.

2.2.1 Layout in traditional tools.

With a scanner generator such as Lex or JLex, the specification consists of a list of regexes, optionally with semantic actions attached:

```
regex1 { return tok(sym.TERM1,yytext()); }
regex2 { return tok(sym.TERM2,yytext()); }
layout_regex { /* No semantic action. */ }
regex3 { ...
```

When the scanner runs on an input, it will check its list of regexes in downward order until it finds one that matches the head of the input. It will then dequeue the matching part of the input (the "lexeme," which in Lex and JLex is stored in the variable `yytext`) and run the

[1] If no such terminal is matched, the scanner will then scan for all terminals to procure information for an error message.

semantic action associated with that regex; it will then scan again at the point in the input immediately following that scan's lexeme.

If the semantic action returns an object, that object (the "token") is added to the stream of tokens being returned to the parser. If no object is returned, the regex is judged to represent what we call *layout* — parts of the input that are not supposed to be visible to the parser and have no meaning thereto. The most common forms of layout are whitespace and comments.

2.2.2 How Copper handles layout.

Copper has a more sophisticated method for specifying and handling layout. While building a parser, Copper keeps track of which terminals have been designated to appear as layout in which contexts, and the sub-scanner for each parser state scans only for the layout that is valid at that location.

Then, each time the parser calls out to the scanner for a new token, it will match as many tokens of layout (series of spaces, comment blocks, *etc.*) as may precede the non-layout token.

2.2.3 How to specify layout in Copper.

2.2.3.1 Grammar layout.

The simplest sort of layout in Copper is *grammar* layout, which should suit the needs of most grammar writers. To designate a terminal as grammar layout, simply place the modifier `ignore` in front of its declaration, as is done on the terminal `WS` in Algorithm 1. This has a similar effect to giving a terminal no semantic action in Lex or JLex, with the important exception that it does not *per se* prevent the parser from using that terminal in a non-layout capacity.

Any number of grammar-wide layout tokens may appear at the beginning and end of the input. It may also appear between any two input tokens, except where explicitly specified otherwise (see section 2.2.3.2).

2.2.3.2 Layout per production.

Copper allows each production to override the grammar-wide layout and specify which terminals may appear as layout between the strings derived from symbols on its right-hand side.

An *empty* layout set may be explicitly specified, as is done on the production `expr ::= UNARY_MINUS expr` in Algorithm 1. In this example, spaces are not permitted between the "negative" sign and the expression it negates (although they are permitted inside the latter).

If instead layout sets contain one or more terminals, they behave similarly to grammar layout in their designated contexts. For example, if the production `expr ::= expr PLUS expr` in Algorithm 1 had a layout terminal with regex `[_]+` (one or more underscores), the

string `3 - 1___+__2` would be valid input, while the string `3 - 1+ 2` would not, because the space between `+` and `2` is not valid layout in that context.

The algorithm calculating what layout is valid where is quite a complex one, but rules-of-thumb when specifying layout are as follows:

- In any context where a terminal from the right-hand side of a certain production can be shifted (except the leftmost one), expect the layout of that production.
- In any context where a nonterminal from the right-hand side of a certain production can be reduced (except the rightmost one), expect the layout of that production.
- Layout specified on productions with zero or one symbols on the right-hand side is meaningless.

For details of how to specify layout per production in Copper, see Section 3.5.3.

2.3 Lexical precedence paradigm.

It is possible for the languages of regexes to overlap, creating ambiguities in which several regexes match a given lexeme. The most common of these is the *keyword-identifier* ambiguity, where a language keyword such as `int` also matches the regex given for identifiers.

In the operation of a scanner from a traditional scanner generator, as described above, no ambiguities are possible because the regex list is gone through one at a time, and the first one that matches is always used. Only if two terminals share the exact same regex is any further kind of disambiguation possible.

Lexical precedence on terminals is here defined as a relation that determines, whenever several terminals have a regex matching a certain lexeme, which terminal should match. The above approach mandates a linear order on terminals: each terminal must take a place on a line, and the terminal closest to the front of the line always matches.

Copper, on the other hand, allows a more generalized lexical precedence relation. Instead of putting terminals in a line, lexical precedence is specified in Copper by individual statements of one of the following forms:

1. "Terminal x has precedence over terminal y," or
2. "If an ambiguity occurs among terminals x, y, *etc.,* return one of them."

Context-aware scanning, with its many sub-scanners scanning for restricted sets of regexes, eliminates most ambiguities and makes this scheme practical.

2.3.1 Dominate/submit-lists.

The primary sort of lexical precedence declarations used in Copper are *dominate-lists* and *submit-lists*, specified on terminals. They implement the first sort of statements listed above.

As the names might suggest, a terminal x's dominate-list is a list of terminals taking precedence over x, while x's submit-list is a list of terminals over which x takes precedence.

Formally, x is on y's submit-list iff y is on x's dominate-list; however, in the actual grammar specifications, one of these will do for both. For details of how to specify these lists in Copper, see Section 3.4.2.

N.B.: The precedence relation created by dominate- and submit-lists is *intransitive*; *i.e.*, if terminal y is on terminal x's submit-list, and z on y's, it does not follow that z is on x's submit-list. z must be placed on that list explicitly in such a case.

N.B.: The precedence relation created by dominate- and submit-lists is *context-insensitive*; *i.e.*, if terminal y is on terminal x's dominate-list, then even in a sub-scanner that is scanning for x but not y, nothing matching y will match x.

2.3.2 Disambiguation functions/groups.

The other kind of lexical precedence declarations in Copper are *disambiguation functions* and *disambiguation groups*.

A disambiguation function is a function (a Java method, in the case of Copper's implementation) specified for a set of terminals, to disambiguate that particular set. It is meant as a second choice if dominate- and submit-lists do not fit the task. Disambiguation functions implement the second sort of statements described above.

A disambiguation function works as follows: if the input to the scanner at a given point matches the regex of more than one terminal (*e.g.,* the group x, y, and z), and this ambiguity is not able to be resolved through dominate- and submit-lists, the scanner will check to see if there has been a disambiguation function for x, y, and z specified. If so, it will execute the function, which takes in the matched lexeme and returns exactly one of x, y, and z.

One use for disambiguation functions is the "typename-identifier" ambiguity occurring when parsing C: typenames and identifiers share a regex; if a name has been defined as a type using a `typedef`, it is scanned as a typename, and otherwise it is scanned as an identifier.

This ambiguity may be resolved with a disambiguation function specified for typenames and identifiers, which returns "typename" if the lexeme is on a list of typenames and "identifier" otherwise.

A *disambiguation group* is a special case of the disambiguation function: instead of a function returning a terminal, it simply specifies the terminal to return. This has the advantage of being declarative.

N.B.: Disambiguation functions and groups are *context-sensitive*; *i.e.*, if there is a disambiguation group on terminals x and y specifying that terminal x should be returned, in a context where only terminal y is valid, y will be matched.

2.4 Transparent prefixes.

The concept of a *transparent prefix* can best be described by example:

Suppose that in some grammar there is a terminal $IntConst$ matching integers and a terminal $FloatConst$ matching floating-point numbers, having the regexes `[0-9]+` and `[0-9]+(\.[0-9]+)?` respectively. Clearly any number without a decimal point matches both, so there is also a disambiguation group on the set $\{IntConst, FloatConst\}$, specifying that $IntConst$ should be returned. In the absence of a decimal point, $IntConst$ will be matched.

Now suppose that there must be some way for the user of the parser to indicate that a number without a decimal point is a floating-point number. This is done using a transparent prefix: a terminal $FloatPrefix$ with regex `float:` is defined, and assigned to be the transparent prefix of $FloatConst$. Now, the integer 214 would be entered as `214`, while the floating-point 214 would be entered as `float:214`. The `float:` prefix is scanned and thrown away like layout (the parser never sees it, hence the *transparent*), but unlike layout, when it is scanned it produces a narrower context that allows $FloatConst$ to be the only valid terminal.

N.B.: Never use transparent prefixes to disambiguate between two terminals that are on each other's dominate- and submit-lists; this does not work due to context-insensitivity.

Chapter 3

The CUP skin.

3.1 Comments and whitespace.

Java-style comments (`//` followed by a comment and a newline, and comments enclosed in `/*` and `*/`) are also recognized as layout in the CUP skin.

N.B.: An exception is in certain contexts in terminal declarations, where line comments are not allowed. See section 3.4.2.

3.2 Preamble.

The *preamble* is a block of Java code that will begin the parser source file to be output. It should contain any needed package or import declarations, as well as any non-public classes to be included in the file.

The preamble is terminated by the string `%%` alone on a line, as shown in Algorithm 1.

3.3 Parser name.

The name of the parser class is provided by a line of the form

```
%parser [classname]
```

occurring on the line directly after the `%%` ending the preamble.

3.4 Lexical syntax blocks.

Lexical syntax blocks are enclosed in the markers

```
%lex{
```

and

```
%lex}
```

They may include any number of declarations of terminals, disambiguation functions, and disambiguation groups.

3.4.1 Terminal class declarations.

For convenience, terminals may be classified into one or more (non-disjoint) sets known as *terminal classes*. Terminal classes are declared with a line of this form:

```
class tclass1[,tclass2,...];
```

Note that such a line only declares the classes, as opposed to specifying which terminals a class contains. That is done in terminal declarations.

3.4.2 Terminal declarations.

The simplest terminal declaration is of this form:

```
terminal [termname] ::= /[regex]/;
```

This declares a terminal with a specified regex that is a member of no terminal classes, does not specify any precedence relations with other terminals (although another terminal may include it on its dominate- or submit-list), and does not have a transparent prefix or semantic action.

A terminal declaration specifying all optional attributes is of this form:

```
ignore terminal [termtype] [termname] ::= /[regex]/

    in ([terminal classes]), < ([submit-list]), > ([dominate-list])
    {:  ...  :} %prefix [prefixname];
```

This declares a terminal that is a member of all terminal classes on the list following `in`, with submit- and dominate-lists containing at least the terminals provided on the lists following < and > respectively, a semantic action returning a designated type, and a transparent prefix.

Submit- and dominate-lists may contain the names of terminal classes as well as the names of terminals. Placing a terminal class on the list is shorthand for placing all the members of that class on the list.

N.B.: Line comments are not allowed to be placed between the `::=` and the regex in terminal declarations, because the double-slash of the line comment is indistinguishable from a regex for the empty string (as in `terminal ws ::= //;`).

Algorithm 3 Terminal declaration example.

```
%lex{
    class keywords;

    terminal INT ::= /int/
        in (keywords), < (), > ();
    terminal FLOAT ::= /float/
        in (keywords), < (), > ();
    /* Return a String: the token's lexeme */
    terminal String IDENTIFIER ::= /[a-z]+/
        in (), < (keywords), > ()
    {:
        RESULT = lexeme;
    :};
%lex}
```

3.4.2.1 Semantic actions on terminals.

Semantic actions on terminals work differently in Copper than in other scanner generators. While in JLex semantic actions are specified on regexes and return an object identifying the matched terminal, as described above, in Copper the semantic action is only run after it is certain what terminal has been matched.

Therefore, the semantic actions of terminals take on an identical format to those of productions in CUP. A variable `RESULT`, of the type specified by `termtype` (default is `Object`) is available inside the semantic action block; what is written to `RESULT` will be returned and is available to access in production semantic actions.

In JLex's semantic actions, the matching lexeme is referred to by the name `yytext`. In Copper, the name `lexeme` is used instead.

Example. Consider a language with two keywords, `INT` and `FLOAT`, and identifiers defined as strings of one or more lowercase letters. This language is defined by the lexical syntax block in Algorithm 3.

3.4.3 Disambiguation functions/groups.

A disambiguation function takes this form:

```
disambiguate [groupname]:(term1,term2[,term3,...])
{:

    [body of Java method returning one of term1, term2, ...]
```

Algorithm 4 Disambiguation group example.

```
%lex{
    terminal INT ::= /int/;
    terminal FLOAT ::= /float/;
    terminal String IDENTIFIER ::= /[a-z]+/
    {:
        RESULT = lexeme;
    :};

    disambiguate ID_FLOAT:(FLOAT,IDENTIFIER) ::= FLOAT;
    disambiguate ID_INT:(INT,IDENTIFIER) ::= INT;
%lex}
```

```
:};
```

A disambiguation group takes this form:

```
disambiguate [groupname]:(term1,term2[,term3,...])

    ::= [one of term1, term2, ...];
```

Example. Consider once again the example from above. As specified there with dominate- and submit-lists, `INT` and `FLOAT` are *reserved* keywords, *i.e.*, they cannot be used as identifiers even in contexts where `INT` and `FLOAT` are invalid syntax.

Suppose that instead the strings `int` and `float` should only be interpreted as keywords in contexts where they are valid, and as identifiers everywhere else. Disambiguation groups may be used to implement this, as shown in Algorithm 4.

3.5 Context-free syntax blocks.

Context-free syntax blocks are enclosed in the markers

```
%cf{
```

and

```
%cf}
```

They may include one declaration of a start symbol, and any number of declarations of nonterminals, operator precedence relations, and productions. With very few exceptions, these take the same form as in CUP.

3.5.1 Nonterminal/start-symbol declarations.

Nonterminal declarations take the familiar form:

```
non terminal [nttype] ntname1[,ntname2,...];
```

This declares one or more grammar nonterminals. If a type (`nttype`) is provided, the variable `RESULT` declared in the semantic action of any production with one of these nonterminals on its left-hand side will be of type `nttype`. If a type is not provided, the default is `Object`.

The declaration of a grammar's start symbol takes the self-explanatory form

```
start with [ntname];
```

3.5.2 Operator precedence/associativity declarations.

Operator precedence and associativity declarations take the familiar form:

```
precedence (left/right/nonassoc) term1[,term2,...];
```

Terminals listed on the same line have identical *operator precedence*, while terminals listed on successive lines have successively higher precedence; *e.g.*, in Algorithm 1, `TIMES` has a higher precedence than `PLUS`, while `PLUS` and `BINARY_MINUS` have equal precedence. All terminals on a line have the *operator associativity* specified on that line.

These precedences and associativities are used to resolve shift-reduce conflicts, using the following logic:

- Operators for the shift and reduce actions are defined:
  - The shift action's operator is the terminal that would be shifted.
  - The reduce action's operator is the operator of the production that would be reduced. This is by default the last terminal on the right-hand side of the production (*e.g.* `+` in `NT ::= NT * NT + NT`), but this can be overridden — see the `%prec` attribute in the next section.)
- If the two operators have different precedence, resolve the conflict in favor of the action whose operator has the highest precedence.
- If the two operators have the same precedence and the same associativity:
  - If the associativity is `left`, resolve in favor of the reduce action.
  - If the associativity is `right`, resolve in favor of the shift action.
  - If the associativity is `nonassoc`, remove both actions — the operator is meant to have its associativity defined through parentheses, or some other manner.
- Otherwise, report the conflict as unresolvable.

3.5.3 Production declarations.

Production declarations take the form:

```
[ntname] ::=

       [sym1[:label1] ...]
        {:
            /* Semantic action for [ntname] ::= RHS1
            */
        :}
        [%prec [termname]] [%layout ([term1,..])]
    [ | RHS2 ...
    ...]
    ;
```

This form is identical in most respects to that used in CUP. The declaration starts with a nonterminal, giving the left-hand side of the productions to follow, followed by `::=`. Then come one or more sequences of zero or more terminals and nonterminals (right-hand sides), separated by vertical bars. Each right-hand side may optionally have a semantic action and two attributes:

- **Custom operator.** As in CUP, adding the attribute

  `%prec [termname]`

  to a production will change the production's operator from the default of the last terminal on the right, to `termname`.

- **Custom layout.** The only bit of production syntax differing from CUP's, this allows specification of custom layout on productions. Adding the attribute

  `%layout (term1,...,termn)`

  to a production will change the layout on that production from the grammar layout set to the set `term1,...,termn`. If an empty layout set is provided, this specifies that no layout may occur between the right-hand side symbols of the production.

3.5.3.1 Semantic actions.

Semantic actions on productions are identical to those in CUP. Any right-hand-side symbols that have been labeled may be accessed inside the semantic action using the label name, as demonstrated in Algorithm 2.

3.6 User code blocks.

3.6.1 Auxiliary.

Auxiliary code is inserted in the body of the parser class. It is meant to hold fields and methods accessed by semantic actions and/or outside classes, such as additional constructors.

An auxiliary code block takes this form:

```
%aux{

    [code block]

%aux}
```

3.6.2 Initialization.

Initialization code is inserted in the body of a method run when the parser is started. It is meant to hold initializations of parser attributes.

An initialization code block takes this form:

```
%init{

    [code block]

%init}
```

3.7 Parser attributes.

A *parser attribute* is a variable meant for use *exclusively* in semantic actions. Unlike fields specified in auxiliary code, parser attributes can be accessed neither from auxiliary code nor from outside classes.

A parser attribute is declared as follows:

```
%attr [attrtype] [attrname];
```

Both type and name are mandatory.

Chapter 4

Running Copper.

4.1 Requirements.

To compile a Copper parser, you need:

- Java Runtime Environment v.1.5 or greater.
- 256MB of memory (512–768 recommended if compiling large grammars).
- `CopperCompiler.jar` on the classpath.

4.2 Command-line interface.

Copper's full command-line syntax is

```
java -jar [location/]CopperCompiler.jar [ -? ]  [ -version ]
[ -package packagename ] [ -parser classname ] [ -o outputfile ]
([ -q ] | [ -v ] | [ -vv ]) [ -mda ] [ -logfile file ]
([ -dump ] | [ -errordump ]) [ -dumpfile file ] [ -dumptype type ]
[ -skin skinname ] [ -engine enginename ] [ -pipeline pipelinename ]
[custom-switches] spec-file1 spec-file2 ...  spec-filen
```

All switches are optional. Most of the parameter-bearing switches have default values for when the switch is omitted.

4.2.1 Quick-start.

The simplest usage of Copper is

```
java -jar [location/]CopperCompiler.jar [ -o parserfile ] specfile
```

This command takes a grammar specification in `specfile`, written in the CUP skin, and compiles it to a parser class of the package and class name specified in the specification itself; if the `-o` switch was specified, the source code of this parser class is output to `parserfile`. The other settings are at defaults.

4.2.2 Switches.

4.2.2.1 `-?`.

Displays the full list of command-line options for Copper, including supported parameters for the `-engine`, `-pipeline`, and `-skin` switches.

`java -jar [location/]CopperCompiler.jar -pipeline P -?` will also list any custom switches available for the pipeline `P`.

4.2.2.2 `-version`.

Displays the Copper version number, the identifier of the source code revision from which the specific JAR was built and the time at which it was built.

4.2.2.3 `-package`.

Specifies what package the output parser should be placed in.

N.B.: Do not specify packages both on the command line and in the specification; this will cause an error when compiling the parser source.

4.2.2.4 `-parser`.

Specifies what the name of the parser class should be. Overrides the name specified by a `%parser` directive.

4.2.2.5 `-o`.

Specifies a file in which to place the output parser class. `-o -` will output the parser class to standard output. Omitting `-o` altogether will result in parser output being suppressed; this option is useful for checking a grammar for conflicts or other errors.

4.2.2.6 `-q`, `-v`, and `-vv`.

By default, after running, Copper outputs a "final report" giving details about the parser, such as the number of parse states, and how many conflicts were located and resolved.

- The `-q` switch turns this off.
- The `-v` switch causes additional compilation information, such as how parse conflicts were resolved, to be output to standard error.
- The `-vv` switch causes much debugging information to be output to standard error. This switch is meant mainly for use by Copper developers.

4.2.2.7 `-mda`.

This runs Copper's modular determinism analysis over a pair of input grammars. The analysis, meant to prove certain properties on language extensions, is not of interest to most grammar writers and does not work with the CUP skin.

4.2.2.8 `-logfile`.

Specifies a file to which standard error should be redirected.

4.2.2.9 `-dump`, `-errordump`, `-dumpfile`, `-dumptype`.

CUP users will be familiar with the `-dump` switch, which tells Copper to output a formatted description ("dump") of a parser, containing the terminals, nonterminals, productions, and disambiguation functions/groups of the implemented grammar, as well as the lexical precedence graph (a matrix combining information from all dominate- and submit-lists), and dumps of the LALR(1) DFA and parse table.

Copper can produce a dump in several different formats, listed below. See section 4.4 for a complete description of the two main dump formats.

- To output a dump, either the `-dump` or the `-errordump` switch must be specified. Using `-errordump` produces a dump only in the event of a parser compilation error (a parse table conflict, for example), while using `-dump` always produces a dump.
- If `-dumpfile` is specified, the dump will be sent to `file` instead of standard error.
- If `-dumptype` is specified, the dump will be of the specified type, instead of plaintext. Available dump types are:
  - `plain`: A plaintext dump adhering as closely as possible to the CUP convention.
  - `html`: An HTML dump following the same general format as the plaintext dump, but thoroughly cross-referenced with symbol names and state numbers.
  - `xmlspec`: Outputs the provided grammar specification in the XML format meant for use with machine-generated specifications. This effectively translates a grammar specified in one of Copper's skins to the XML schema.

4.2.2.10 `-pipeline`.

Copper is designed to support several compilation "pipelines," or routines for compilation. These may compile using different algorithms or implementations, or to different output formats.

There are two pipelines currently supported in Copper, `default` (a practically-oriented compiler implementation introduced in Copper 0.7) and `legacy` (the research-oriented implementation from versions 0.6 and earlier, which supports the experimental features of Copper).

Unless grammar writers are using the experimental features of Copper, they should use the `default` pipeline.

4.2.2.11 `-skin`, `-engine`.

The `default` and `legacy` pipelines both divide compilation into three phases: parse the input spec, compile it into an LR DFA and scanner, and output the resulting Java class.

On the input side, as mentioned, there are several input formats or "skins" that may be used interchangeably for the input-spec-parsing phase. Most grammar writers will use the default skin, `cup`. The XML schema is also provided as a skin, `xml`. Additionally, one can specify a grammar using the Java API and pass it directly to the compiler without parsing (see the method `ParserCompiler.compile(ParserBean,ParserCompilerParameters)` in Copper's Javadoc).

On the output side, there are different "engines," or parser implementations, into which the compiled LR DFA can be placed. Most grammar writers will use the default `single` engine, which is the only engine supported outside the `legacy` pipeline.

4.3 Copper ANT task and API.

In addition to the command-line interface, Copper provides an ANT task, named `edu.umn.cs.melt.copper.ant.CopperAntTask`, and an API in which parameters (and, optionally, parser specifications themselves) are specified as Java bean objects. Both are fully documented in Copper's Javadoc.

4.4 Format of the Copper grammar dump.

4.4.1 Text dump.

This dump format adheres as closely as possible to its CUP analogue.

4.4.1.1 Terminals, nonterminals, productions, LALR(1) DFA.

These are put forth exactly as in CUP. Grammar constructs appear in lists with each one getting a number that appears in brackets immediately before it. DFA states appear as lists of LR(1) items followed by transitions.

4.4.1.2 Lexical precedence graph.

The precedence graph output is a representation of the precedence relations among the terminal set. It is in the DOT graph format and may be input to a tool such as *Graphviz* for viewing.

The vertices of the graph are labeled with one or more terminal numbers. These represent groups of terminals whose dominate-lists are all equal and whose submit-lists are all equal; *e.g.*, a group of keywords with the same identifier in their dominate-lists. An arrow

pointing from vertex x to vertex y means that all the terminals in vertex x take precedence over all the terminals in vertex y.

4.4.1.3 Disambiguation functions/groups.

Each disambiguation function or group is given a number, which appears in brackets immediately before it.

A disambiguation function or group is identified by its name and the set of terminals it disambiguates: `name : {term1,term2,...}`. If it is a disambiguation group, this will be suffixed with `-> termi`, where `termi` is the target of the disambiguation group.

4.4.1.4 Parse table.

The format of this dump is identical to CUP's with regard to all constructs that appear in parse tables made by that parser generator. Copper adds more information to each state about layout and transparent prefixes:

```
[layout term X]
[prefix term Y -> terms Z,A,...]
```

Constructs of this form mean that terminal `X` is valid as layout in the given state, while `Y` is valid as a transparent prefix and terminals `Z`, `A`, *etc.* may validly occur following `Y`.

4.4.2 HTML dump.

The HTML dump has the same basic layout as the text dump, but is thoroughly cross-referenced via hyperlinks. It also omits the lexical precedence graph in favor of displaying the submit- and dominate-lists for each terminal.

4.5 Grammar troubleshooting.

In this section, five problems encountered when compiling grammars in Copper are discussed.

4.5.1 Heap overflow.

On larger grammars, Copper's memory requirements sometimes exceed the JVM's default maximum heap size, at which point compilation will terminate with an `OutOfMemoryError`. If this occurs, allocate more memory to the JVM by increasing the maximum heap size with the `-Xmx` switch, *e.g.*:

```
java -Xmx1024m -jar ...
```

The `-Xmx` switch is "nonstandard and subject to change without notice."

4.5.2 "Cyclic precedence relation involving terminals ..."

On occasion Copper will give an error of this form:

```
Cyclic precedence relation involving terminals
[...,]
```

This means that (1) there is a cyclic precedence relation among the listed terminals (*i.e.* there is no way to say that one of the terminals has the *maximum* precedence) and (2) they can all occur in the same context.

4.5.3 Parse table conflict.

As in a traditional parser generator, a parse table conflict occurs when two actions are placed in the same cell of the LR parse table; such a conflict is usually resolved by specifying precedence and associativity on terminals.

Unlike a traditional parser generator, however, Copper does not make any attempt to resolve such conflicts automatically. Using the CUP skin, reduce-reduce conflicts are automatically resolved by the order in which conflicting productions appear in the file, as is done in CUP. Any shift-reduce conflicts are reported as compilation errors.

The `default` pipeline suppresses output of parser code , but the `legacy` pipeline may not. Undefined behavior occurs if a parser compiled from a conflicting table is run.

4.5.4 Lexical ambiguity.

Copper is able to guarantee that there is no lexical ambiguity in its scanners, if certain compile-time checks pass. When any such checks fail, it is reported as a compilation error, of this form:

```
Unresolvable lexical ambiguity at parser states
    [...] (between/among) terminals:
        [...,
        ...]
```

This means that the the set of terminals given are not on each other's dominate- or submit-lists, and there is no disambiguation function or group assigned to the set. There are three ways to resolve the ambiguity:

- Modify the dominate- and submit-lists of the set of terminals;
- Add a disambiguation function or group to disambiguate the set appropriately;
- Alter the context-free syntax so this set of terminals do not appear in the same context.

Chapter 5

Running a Copper parser.

5.1 Requirements.

To run a Copper parser, you need:

- Java Runtime Environment v.1.5 or greater.
- `CopperRuntime.jar` or `CopperCompiler.jar` on the classpath.

5.2 Constructors.

No parameters need to be passed to a Copper parser on construction. It is possible to specify additional constructors in the auxiliary code; however, the constructor with no parameters cannot be specified in that manner.

5.3 `parse()` methods.

A parser has several methods nameed `parse` that may be used to run the parser. These methods are documented in Copper's Javadoc for the interface `edu.umn.cs.melt.copper.runtime.engines.CopperParser`.

- `parse(Reader input)`
- `parse(Reader input,String inputName)`
- `parse(String text)`
- `parse(String text,String inputName)`

Each function returns an `Object` that is the `RESULT` of the last production reduced in the parse (*i.e.*, the root of the parse tree).

The arguments to the methods are as follows.

- `text` is a string containing text to parse.

- `input` is a `Reader` containing text to parse.
- `inputName` (defaults to "`<stdin>`") is a label for `input` or `text`.

5.4 The class `RunParser`.

The Copper parser runtime includes a runnable class `edu.umn.cs.melt.copper.runtime.RunParser` that will call into a Copper parser's `parse` method from the command line. If the parser class being run specifies a method `runPostParseCode(Object)`, it will then call that method on the returned parse object.

Its full command-line syntax is:

```
java -classpath [location/]CopperRuntime.jar:[parser-classpath]
      edu.umn.cs.melt.copper.runtime.RunParser
      parser-class-name [-v] [-f input-file]
```

- `-f` tells the parser to read from the given `input-file` instead of from standard input.
- `-v` tells the parser to output the full stack trace of any exceptions thrown, instead of just their message.

Appendix A

CUP skin grammar.

A.1 Lexical syntax.

```
assoctypes_kwd ::= left | right | nonassoc
name_tok ::= [A-Za-z_][A-Za-z0-9_]*
prec_number ::= 0|([1-9][0-9]*)
character ::= .
termname ::= [^\:]+
escaped ::= \\.
code_t ::= ([^%:]*|[^%\n]%|:[^}])*
ws ::= ([ \t\n]+)|(//.*)|(/\*([^*]|\*[^/])*\*/)
ws_no_line ::= ([ \t\n]+)|(/\*([^*]|\*[^/])*\*/)
```

A.2 Context-free syntax.

A.2.1 Specification syntax.

All Java layout (whitespace, block and line comments) is allowed between the symbols of these productions.

```
GrammarFile ::=
    code_t '%%' ParserDecl DeclBlocks
ParserDecl ::=
    '%parser' name_tok
DeclBlocks ::=
      DeclBlock DeclBlocks
    | DeclBlock
DeclBlock ::=
      '%attr' TypeName name_tok ';'
    | '%aux{' code_t '%aux}'
```

```
    | '%init{' code_t '%init}'
    | '%lex{' LexDecls '%lex}'
    | '%cf{' CFDecls '%cf}'

LexDecls ::=

    LexDecl LexDecls
    | LexDecl

CFDecls ::=

    CFDecl CFDecls
    | CFDecl

LexDecl ::=

    'class' CommaSymSeq ';'
    | IgnoreOpt 'terminal' name_tok SuperRegexRoot
       PrecListsOpt CodeBlockOpt TerminalFlags ';'
    | IgnoreOpt 'terminal' TypeName name_tok '::='
       RegexRoot PrecListsOpt CodeBlockOpt TerminalFlags ';'

PrecLists ::=

    PrecList ',' PrecLists
    | PrecList

PrecList ::=

    'in' '(' CommaSymSeqOpt ')'
    | '<' '(' CommaSymSeqOpt ')'
    | '>' '(' CommaSymSeqOpt ')'

PrecListsOpt ::=

    PrecLists |

TerminalFlags ::=

    '%prefix' name_tok TerminalFlags
    |

LexDecl ::=

    'disambiguate' name_tok ':' '(' CommaSymSeq ')'
       '{:' code_t ':}' ';'
    | 'disambiguate' name_tok ':' '(' CommaSymSeq ')'
       '::=' name_tok ';'

CodeBlockOpt ::=

    '{:' code_t ':}' |

IgnoreOpt ::=

    'ignore' |

CFDecl ::=
```

```
    'precedence' assoctypes_kwd CommaOrSymSeq ';'

CommaOrSymSeq ::=

    name_tok SymSeq
    | name_tok ',' CommaSymSeq

CFDecl ::=

    'non' 'terminal' TypeName CommaSymSeq ';'
    | 'non' 'terminal' CommaSymSeq ';'

TypeNameOpt ::=

    TypeName
    |

TypeName ::=

    QualifiedName
    | QualifiedName '<' TypeNameSeq '>'

QualifiedName ::=

    name_tok
    | name_tok wildcard QualifiedName

TypeNameSeq ::=

    TypeName ',' TypeNameSeq
    | TypeName

CFDecl ::=

    'start' 'with' name_tok ';'
    | name_tok '::=' RHSSeq ';'

RHS ::=

    LabeledSymSeq CodeBlockOpt RHSFlags

RHSSeq ::=

    RHS '|' RHSSeq
    | RHS

RHSFlags ::=

    '%prec' name_tok RHSFlags
    | '%layout' '(' CommaSymSeqOpt ')' RHSFlags
    |

CommaSymSeqOpt ::=

    CommaSymSeq |

CommaSymSeq ::=

    name_tok ',' CommaSymSeq
    | name_tok
```

```
LabeledSymSeq ::=
    name_tok ':' name_tok LabeledSymSeq
    | name_tok LabeledSymSeq
    |
SymSeq ::=
    name_tok SymSeq
    |
```

A.2.2 Regex bridge syntax.

Encompasses the context immediately before the specification of a terminal's regex. Only whitespace and block comments are allowed here.

```
SuperRegexRoot ::=
    '::=' RegexRoot %layout (ws_no_line)
```

A.2.3 Regex syntax.

No whitespace of any sort is allowed between the symbols of these productions.

```
RegexRoot ::=
    '/' Regex_R '/'
    | '/' '/'
Regex_R ::=
    Regex_DR
    | Regex_DR '|' Regex_R
Regex_DR ::=
    Regex_UR Regex_RR
    | Regex_UR '*' Regex_RR
    | Regex_UR '+' Regex_RR
    | Regex_UR '?' Regex_RR
Regex_RR ::=
    Regex_DR
    |
Regex_UR ::=
    Regex_CHAR
    | '.'
    | '[' Regex_G ']'
    | '[' '^' Regex_G ']'
    | '[' ':' termname ':' ']'
    | '(' Regex_R ')'
```

```
Regex_G ::=
    Regex_UG Regex_RG
Regex_UG ::=
    Regex_CHAR
    | Regex_CHAR '-' Regex_CHAR
Regex_RG ::=
    Regex_G
    |
Regex_CHAR ::=
    character
    | escaped
```

Appendix B

“Mini-Java” example specification.

```
package ancillary;
%%
%parser MiniJavaRecognizer
%lex{
    class keyword;
    ignore terminal WS ::= /[ \t\n\r]+/;
    ignore terminal LINE_COMMENT ::= /\/\/.*/;
    ignore terminal BLOCK_COMMENT ::= /\/\*([^*]|\*[^/])\*\//;
    terminal ID ::= /[A-Za-z][A-Za-z0-9_]*/
        in (), < (keyword), > ();
    terminal INTEGER ::= /0|([1-9][0-9]*)/;
    terminal CLASS ::= /class/ in (keyword), < (), > ();
    terminal PUBLIC ::= /public/ in (keyword), < (), > ();
    terminal STATIC ::= /static/ in (keyword), < (), > ();
    terminal VOID ::= /void/ in (keyword), < (), > ();
    terminal MAIN ::= /main/ in (keyword), < (), > ();
    terminal STRING ::= /String/ in (keyword), < (), > ();
    terminal EXTENDS ::= /extends/ in (keyword), < (), > ();
    terminal RETURN ::= /return/ in (keyword), < (), > ();
    terminal INT ::= /int/ in (keyword), < (), > ();
    terminal BOOLEAN ::= /boolean/ in (keyword), < (), > ();
    terminal IF ::= /if/ in (keyword), < (), > ();
    terminal ELSE ::= /else/ in (keyword), < (), > ();
    terminal WHILE ::= /while/ in (keyword), < (), > ();
    terminal PRINT ::= /System\.out\.println/
        in (keyword), < (), > ();
    terminal LENGTH ::= /length/ in (keyword), < (), > ();
    terminal TRUE ::= /true/ in (keyword), < (), > ();
    terminal FALSE ::= /false/ in (keyword), < (), > ();
    terminal THIS ::= /this/ in (keyword), < (), > ();
    terminal NEW ::= /new/ in (keyword), < (), > ();
```

```
    terminal LBRACK ::= /\{/ in (keyword), < (), > ();
    terminal RBRACK ::= /\}/ in (keyword), < (), > ();
    terminal LBRACE ::= /\[/ in (keyword), < (), > ();
    terminal RBRACE ::= /\]/ in (keyword), < (), > ();
    terminal LPAREN ::= /\(/ in (keyword), < (), > ();
    terminal RPAREN ::= /\)/ in (keyword), < (), > ();
    terminal DOT ::= /\./ in (keyword), < (), > ();
    terminal PLUS ::= /\+/ in (keyword), < (), > ();
    terminal MINUS ::= /-/ in (keyword), < (), > ();
    terminal TIMES ::= /\*/ in (keyword), < (), > ();
    terminal AND ::= /&&/ in (keyword), < (), > ();
    terminal LT ::= /</ in (keyword), < (), > ();
    terminal NOT ::= /!/ in (keyword), < (), > ();
    terminal ASSIGN ::= /=/ in (keyword), < (), > ();
    terminal SEMICOLON ::= /;/ in (keyword), < (), > ();
    terminal COMMA ::= /:/ in (keyword), < (), > ();

%lex}
%cf{

    non terminal classdecl;
    non terminal classdecls;
    non terminal exp;
    non terminal exprest;
    non terminal exprests;
    non terminal explist;
    non terminal formallist;
    non terminal formalrests;
    non terminal formalrest;
    non terminal mainclass;
    non terminal methoddecl;
    non terminal vardeclsandstmts;
    non terminal methoddecls;
    non terminal program;
    non terminal nonidstmt;
    non terminal stmt;
    non terminal stmts;
    non terminal nonidtype;
    non terminal type;
    non terminal vardecl;
    non terminal vardecls;
    non terminal mainstuff;

    precedence left AND;
    precedence left LT;
    precedence left PLUS,MINUS;
```

```
precedence left TIMES;
precedence right NOT;
precedence left DOT,LBRACK;

start with program;

program ::=
      mainclass:mc classdecls:cdls
    ;
mainclass ::=
      CLASS:c ID:name mainstuff ID:arg RPAREN
       LBRACE stmt:body RBRACE RBRACE
    ;
mainstuff ::=
      LBRACE PUBLIC STATIC VOID MAIN
       LPAREN STRING LBRACK RBRACK
    ;
classdecl ::=
      CLASS:c ID:name LBRACE vardecls:vlist
       methoddecls:mlist RBRACE
    | CLASS:c ID:name EXTENDS ID:sname LBRACE

       vardecls:vlist methoddecls:mlist RBRACE
    ;
classdecls ::=
      classdecl:c classdecls:clist
    |
    ;
vardecl ::=
      type:t ID:i SEMICOLON
    ;
vardecls ::=
      vardecl:v vardecls:vlist
    |
    ;
vardeclsandstmts ::=

    nonidtype:t ID:name SEMICOLON vardeclsandstmts:rest
    | nonidstmt:s stmts:stmlist
    | ID:t ID:name SEMICOLON vardeclsandstmts:rest
    | ID:var ASSIGN exp:e SEMICOLON stmts:stmlist
```

```
    | ID:var LBRACK exp:e1 RBRACK ASSIGN exp:e2

       SEMICOLON stmts:stmlist
    |
    ;
methoddecl ::=

    PUBLIC:p type:t ID:name LPAREN formallist:fl RPAREN
       LBRACE vardeclsandstmts:rest RETURN exp:e
       SEMICOLON RBRACE
    ;
methoddecls ::=
      methoddecl:m methoddecls:rest
    |
    ;
formallist ::=
      type:t ID:name formalrests:rest
    |
    ;
formalrest ::=
      COMMA type:t ID:name
    ;
formalrests ::=
      formalrest:f formalrests:rest
    |
    ;
type ::=
      nonidtype:t
    | ID:i
    ;
nonidtype ::=
      INT:i
    | INT:i LBRACK RBRACK
    | BOOLEAN:b
    ;
stmt ::=
      nonidstmt:s
    | ID:var ASSIGN exp:e SEMICOLON
    | ID:var LBRACK exp:e1 RBRACK ASSIGN exp:e2 SEMICOLON
    ;
stmts ::=
```

```
      stmt:s stmts:rest
    |
    ;
nonidstmt ::=
      LBRACE stmts:rest RBRACE
    | IF:start LPAREN exp:e RPAREN stmt:s1 ELSE stmt:s2
    | WHILE:start LPAREN exp:e RPAREN stmt:s
    | PRINT:start LPAREN exp:e RPAREN SEMICOLON
    ;
exp ::=
      exp:e1 AND exp:e2
    | exp:e1 LT exp:e2
    | exp:e1 PLUS exp:e2
    | exp:e1 MINUS exp:e2
    | exp:e1 TIMES exp:e2
    | exp:e1 LBRACK exp:e2 RBRACK
    | exp:array DOT LENGTH
    | exp:obj DOT ID:name LPAREN explist:fl RPAREN
    | INTEGER:i
    | TRUE:t
    | FALSE:f
    | ID:i
    | THIS:t
    | NEW:start INT LBRACK exp:e RBRACK
    | NEW:start ID:name LPAREN RPAREN
    | NOT:start exp:e
    | LPAREN:start exp:e RPAREN
    ;
explist ::=
      exp:e exprests:rest
    |
    ;
exprest ::=
      COMMA exp:e
    ;
exprests ::=
      exprest:e exprests:rest
    |
    ;
%cf}
```

Appendix C

License.