Chapter 4 Parsing Using Java CUP

CUP is a parser generator. It takes a CUP program - essentially an LALR(1) parsable grammar, and generates a Java program that will parse input that satisfies that grammar.

CUP assumes that a lexical analyser is provided separately. Normally, the lexical analyser is generated using JFlex. Additional support code can be provided in Java.

An example (INTERP1)

The following JFlex, CUP, and Java source files together generate a simple "interpreter". This interpreter reads lines of input, parses them, and "interprets" them. A line of input is either an assignment statement, that specifies the value of a variable, or an expression to be printed out. In the first case, the interpreter evaluates the expression, and stores the value in the variable. In the second case the interpreter evaluates the expression, and prints out the value.

The lexical analyser

```
package grammar;

import java.io.*;
import java_cup.runtime.*;

%%

%public
%type         Symbol
%char

%{
     public Symbol token( int tokenType ) {
          System.err.println(
               "Obtain token " + sym.terminal_name( tokenType )
               + " \"" + yytext() + "\"" );
          return new Symbol( tokenType, yychar,
               yychar + yytext().length(), yytext() );
          }

%}

number        =    [0-9]+
ident         =    [A-Za-z][A-Za-z0-9]*
space         =    [\ \t]
newline       =    \r|\n|\r\n

%%

"="           { return token( sym.ASSIGN ); }
"+"           { return token( sym.PLUS ); }
"-"           { return token( sym.MINUS ); }
"*"           { return token( sym.TIMES ); }
"/"           { return token( sym.DIVIDE ); }
"("           { return token( sym.LEFT ); }
")"           { return token( sym.RIGHT ); }
{newline}     { return token( sym.NEWLINE ); }
{space}       { }

{number}      { return token( sym.NUMBER ); }
{ident}       { return token( sym.IDENT ); }
```

```
.               { return token( sym.error ); }
<<EOF>>         { return token( sym.EOF ); }
```

The lexical analyser matches special symbols “=”, “+”, “-”, “/”, “(”, “)”, integer constants, and identifiers. In this case, newlines are syntactically important, but blanks and tabs are not, so I return a token for newlines, but not for blanks and tabs. If there is a lexical error, I return an error token (that generates a syntax error for the parser).

The parser

```
package grammar;

import java.util.*;
import java.io.*;
import java_cup.runtime.*;

action code
     {:
     Hashtable table = new Hashtable();
     :};

parser code
     {:
     private Yylex lexer;
     private File file;

     public parser( File file ) {
          this();
          this.file = file;
          try {
               lexer = new Yylex( new FileReader( file ) );
               }
          catch ( IOException exception ) {
               throw new Error( "Unable to open file \"" + file + "\"" );
               }
          }
...
     :};

scan with
     {:
     return  lexer.yylex();
     :};

terminal LEFT, RIGHT, NEWLINE, PLUS, MINUS, TIMES, DIVIDE, ASSIGN;
terminal String NUMBER;
terminal String IDENT;

nonterminal StmtList, Stmt;
nonterminal Integer Expr, Term, Factor;

start with StmtList;

StmtList::=

     |
          StmtList Stmt
     ;
```

```
Stmt::=
          IDENT:ident ASSIGN Expr:expr NEWLINE
          {:
          table.put( ident, expr );
          :}
     |
          Expr:expr NEWLINE
          {:
          System.out.println( expr.intValue() );
          :}
     |
          error NEWLINE
     |
          NEWLINE
     ;

Expr::=
          Expr:expr PLUS Term:term
          {:
          RESULT = new Integer( expr.intValue() + term.intValue() );
          :}
     |
          Expr:expr MINUS Term:term
          {:
          RESULT = new Integer( expr.intValue() - term.intValue() );
          :}
     |
          MINUS Term:term
          {:
          RESULT = new Integer( - term.intValue() );
          :}
     |
          Term:term
          {:
          RESULT = term;
          :}
     ;

Term::=
          Term:term TIMES Factor:factor
          {:
          RESULT = new Integer( term.intValue() * factor.intValue() );
          :}
     |
          Term:term DIVIDE Factor:factor
          {:
          RESULT = new Integer( term.intValue() / factor.intValue() );
          :}
     |
          Factor:factor
          {:
          RESULT = factor;
          :}
     ;

Factor::=
          LEFT Expr:expr RIGHT
          {:
          RESULT = expr;
          :}
     |
```

```
          NUMBER:value
          {:
          RESULT = new Integer( value );
          :}
     |
          IDENT:ident
          {:
          Integer value = ( Integer ) table.get( ident );
          if ( value == null ) {
               parser.report_error( "Undeclared Identifier " + ident,
               new Symbol( sym.IDENT, identleft, identright, ident ) );
               value = new Integer( 0 );
               }
          RESULT = value;
          :}
     ;
```

The lines

```
action code
     {:
     Hashtable table = new Hashtable();
     :};
```

add code to a private action class used to contain the code for the actions. In this case I have added a hash table variable representing a "symbol table" to perform the mapping of identifiers to values. (Isn't it wonderful to not have to program such things!)

The lines

```
parser code
     {:
     private Yylex lexer;
     private File file;

     public parser( File file ) {
          this();
          this.file = file;
          try {
               lexer = new Yylex( new FileReader( file ) );
               }
          catch ( IOException exception ) {
               throw new Error( "Unable to open file \"" + file + "\"" );
               }
          }
...
     :};
```

add code to the parser class. In this case I have added a constructor, and instance field variables representing the lexical analyser and input file.

The lines

```
scan with
     {:
     return  lexer.yylex();
     :};
```

specify that to obtain a token, the parser should invoke lexer.yylex(). Thus there is no obligation to use JFlex to generate the lexical analyser, although it is usually a good idea.

The lines

```
terminal LEFT, RIGHT, NEWLINE, PLUS, MINUS, TIMES, DIVIDE, ASSIGN;
```

```
terminal String NUMBER;
terminal String IDENT;
```

specify the names of the terminal symbols in the grammar. The terminal symbols LEFT, RIGHT, NEWLINE, PLUS, MINUS, TIMES, DIVIDE, ASSIGN have no associated value. NUMBER and IDENT have a String as a value.

The lines

```
nonterminal StmtList, Stmt;
nonterminal Integer Expr, Term, Factor;
```

specify the names of the nonterminal symbols in the grammar. When we match a construct corresponding to StmtList or Stmt, we do not return a value. When we match a construct corresponding to Expr, Term or Factor, we return a value of type Integer. Values must be Objects, not of a primitive type.

The line

```
start with StmtList;
```

specifies that the start symbol is the nonterminal StmtList.

CUP generates a class called sym, which contains definitions of the terminal and nonterminal symbols as integer constants.

```
/** CUP generated class containing symbol constants. */
public class sym {
  /* terminals */
  static final int EOF = 0;
  static final int LEFT = 2;
  static final int DIVIDE = 9;
  static final int NEWLINE = 4;
  static final int NUMBER = 11;
  static final int error = 1;
  static final int MINUS = 7;
  static final int TIMES = 8;
  static final int ASSIGN = 10;
  static final int IDENT = 12;
  static final int PLUS = 6;
  static final int RIGHT = 3;

  /* nonterminals */
  static final int $START = 0;
  static final int StmtList = 1;
  static final int Factor = 5;
  static final int Expr = 3;
  static final int Stmt = 2;
  static final int Term = 4;
  }
```

I don't know why it prints them out in such a strange ordering, but I assume it is because it puts the terminal and nonterminal symbols in a hash table, and the order of enumeration depends on their hash codes. The order is different for different implementations of Java. Presumably the difference is that the hashing function for strings are different for different implementations of Java.

The remaining lines specify the grammar of the input to be matched, together with actions to be performed, when a reduction occurs. If we remove the additional code to do with the actions, we have the basic grammar:

```
StmtList::=

    |
        StmtList Stmt
```

```
    ;

Stmt::=
        IDENT ASSIGN Expr NEWLINE
    |
        Expr NEWLINE
    |
        error NEWLINE
    |
        NEWLINE
    ;

Expr::=
        Expr PLUS Term
    |
        Expr MINUS Term
    |
        MINUS Term
    |
        Term
    ;

Term::=
        Term TIMES Factor
    |
        Term DIVIDE Factor
    |
        Factor
    ;

Factor::=
        LEFT Expr RIGHT
    |
        NUMBER
    |
        IDENT
    ;
```

A program is a list of statements terminated by newlines. A statement can be either a null statement (for people who like blank lines), an assignment statement of the form IDENT "=" Expr or just an Expr (expression). An expression can involve terms combined with addition, subtraction, and negation operators. A term can involve factors combined with multiplication and division operators. A factor can be a parenthesised expression, or a number or identifier. Note that I have given negation the same precedence as addition and subtraction, as occurs in mathematics. Most modern computer languages give negation and other prefix operators a higher precedence than infix operators.

The rule Stmt ::= error NEWLINE is used to tell the parser what to do if the input doesn't match the grammar. Essentially it says that if we get a syntax error, cut the stack back until the point where we started to parse the Stmt, push a special error symbol onto the stack, then discard tokens until we get a NEWLINE.

Now, as well as parsing our input, we want to compute the value of the expression, and either assign it to a variable (in the case of an assignment statement), or print out the value (in the case of an expression by itself). Thus we need to add actions to our grammar definition.

Every construct being parsed is given an associated value. Terminal symbols receive their value from the lexical analyser (the value field of the token). For nonterminals, the value is

specified in the action associated with the rule. An action is written after the right hand side of the rule, and is essentially Java code, enclosed in {: ... :}.

We have to have a way of referring to the value associated with a symbol on the right hand side of the rule. To achieve this, we label the symbol by appending a ":" and an identifier, and use this identifier in the action to refer to the value. We also have to have a way of specifying the value associated with the symbol on the left hand side of the rule. To achieve this, we assign to the variable RESULT.

For example, consider the following code:

```
Expr::=
          Expr:expr PLUS Term:term
          {:
          RESULT = new Integer( expr.intValue() + term.intValue() );
          :}
     |
          Expr:expr MINUS Term:term
          {:
          RESULT = new Integer( expr.intValue() - term.intValue() );
          :}
     |
          MINUS Term:term
          {:
          RESULT = new Integer( - term.intValue() );
          :}
     |
          Term:term
          {:
          RESULT = term;
          :}
     ;
```

The value of a construct of the form expr "+" term is the value of expr, plus the value of term, with all values stored in objects of type Integer.

Sometimes we don't want to specify a value, but want to execute code for side effects.

```
Stmt::=
          IDENT:ident ASSIGN Expr:expr
          {:
          table.put( ident, expr );
          :}
     |
          Expr:expr
          {:
          System.out.println( expr.intValue() );
          :}
     ;
```

Thus if we match ident "=" expr, we evaluate Expr, and store the value in the hash table, with the identifier as the key. If we have only an expr, then we print its value out.

The main program

As before, we need code to start things up. We invoke a constructor for the parser class to create a parser. We then perform parsing using this parser. I have given main() a directory name as a parameter. The directory contains an input file. Error and output files are generated in this directory.

```
import grammar.*;
import java.io.*;
import java_cup.runtime.*;
```

```
public class Main {

     public static void main( String[] argv ) {
          String dirName = null;

          try {
               for ( int i = 0; i < argv.length; i++ ) {
                    if ( argv[ i ].equals( "-dir" ) ) {
                         i++;
                         if ( i >= argv.length )
                              throw new Error( "Missing directory name" );
                         dirName = argv[ i ];
                         }
                    else {
                         throw new Error(
                              "Usage: java Main -dir directory" );
                         }
                    }

               if ( dirName == null )
                    throw new Error( "Directory not specified" );

               System.setErr( new PrintStream( new FileOutputStream(
                    new File( dirName, "program.err" ) ) ) );
               System.setOut( new PrintStream( new FileOutputStream(
                    new File( dirName, "program.out" ) ) ) );

               parser p = new parser( new File( dirName, "program.in" ) );
               p.parse();
               //   p.debug_parse();   //   For debugging
               }
          catch ( Exception e ) {
               System.err.println( "Exception at " );
               e.printStackTrace();
               }
          }
     }
```

The parse() method invokes the CUP runtime parser. This is a method that performs an LALR(1) parse, invoking the code within “scan with {: ... :}” whenever it needs to get a token. The parse() method returns an object of type Symbol, composed of the value returned by the start symbol, together with its type and left and right positions.

There is also a method called debug_parse() that not only performs a parse, but also generates debugging messages to the standard error file. It provides a good way of determining what is going on, if there is something wrong with our grammar.

For example, if we had the input

```
a = 3
b = 4
a * a + b * b
```

Then debug_parse() would generate output like

```
# Initializing parser
Obtain token IDENT "a"
# Current token is IDENT
# Reduce by rule StmtList ::=
# Shift nonterminal StmtList to push state #1
# Shift token IDENT to push state #3
Obtain token ASSIGN "="
# Current token is ASSIGN
```

```
# Shift token ASSIGN to push state #27
Obtain token NUMBER "3"
# Current token is NUMBER
# Shift token NUMBER to push state #2
Obtain token NEWLINE "
"
# Current token is NEWLINE
# Reduce by rule Factor ::= NUMBER
# Shift nonterminal Factor to push state #6
# Reduce by rule Term ::= Factor
# Shift nonterminal Term to push state #7
# Reduce by rule Expr ::= Term
# Shift nonterminal Expr to push state #28
# Shift token NEWLINE to push state #29
Obtain token IDENT "b"
# Current token is IDENT
# Reduce by rule Stmt ::= IDENT ASSIGN Expr NEWLINE
# Shift nonterminal Stmt to push state #10
# Reduce by rule StmtList ::= StmtList Stmt
# Shift nonterminal StmtList to push state #1
# Shift token IDENT to push state #3
Obtain token ASSIGN "="
# Current token is ASSIGN
# Shift token ASSIGN to push state #27
Obtain token NUMBER "4"
# Current token is NUMBER
# Shift token NUMBER to push state #2
Obtain token NEWLINE "
"
# Current token is NEWLINE
# Reduce by rule Factor ::= NUMBER
# Shift nonterminal Factor to push state #6
# Reduce by rule Term ::= Factor
# Shift nonterminal Term to push state #7
# Reduce by rule Expr ::= Term
# Shift nonterminal Expr to push state #28
# Shift token NEWLINE to push state #29
Obtain token IDENT "a"
# Current token is IDENT
# Reduce by rule Stmt ::= IDENT ASSIGN Expr NEWLINE
# Shift nonterminal Stmt to push state #10
# Reduce by rule StmtList ::= StmtList Stmt
# Shift nonterminal StmtList to push state #1
# Shift token IDENT to push state #3
Obtain token TIMES "*"
# Current token is TIMES
# Reduce by rule Factor ::= IDENT
# Shift nonterminal Factor to push state #6
# Reduce by rule Term ::= Factor
# Shift nonterminal Term to push state #7
# Shift token TIMES to push state #16
Obtain token IDENT "a"
# Current token is IDENT
# Shift token IDENT to push state #14
Obtain token PLUS "+"
# Current token is PLUS
# Reduce by rule Factor ::= IDENT
# Shift nonterminal Factor to push state #19
# Reduce by rule Term ::= Term TIMES Factor
# Shift nonterminal Term to push state #7
# Reduce by rule Expr ::= Term
```

```
# Shift nonterminal Expr to push state #5
# Shift token PLUS to push state #22
Obtain token IDENT "b"
# Current token is IDENT
# Shift token IDENT to push state #14
Obtain token TIMES "*"
# Current token is TIMES
# Reduce by rule Factor ::= IDENT
# Shift nonterminal Factor to push state #6
# Reduce by rule Term ::= Factor
# Shift nonterminal Term to push state #24
# Shift token TIMES to push state #16
Obtain token IDENT "b"
# Current token is IDENT
# Shift token IDENT to push state #14
Obtain token NEWLINE "
"
# Current token is NEWLINE
# Reduce by rule Factor ::= IDENT
# Shift nonterminal Factor to push state #19
# Reduce by rule Term ::= Term TIMES Factor
# Shift nonterminal Term to push state #24
# Reduce by rule Expr ::= Expr PLUS Term
# Shift nonterminal Expr to push state #5
# Shift token NEWLINE to push state #26
Obtain token EOF ""
# Current token is EOF
# Reduce by rule Stmt ::= Expr NEWLINE
# Shift nonterminal Stmt to push state #10
# Reduce by rule StmtList ::= StmtList Stmt
# Shift nonterminal StmtList to push state #1
# Shift token EOF to push state #11
Obtain token EOF ""
# Current token is EOF
# Reduce by rule $START ::= StmtList EOF
# Shift nonterminal $START to push state #-1
```

CUP Specification Syntax

This material is taken from the CUP manual.

A specification consists of:

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

Each of these parts must appear in the order indicated.

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 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, or underscore (_), which can then be followed by zero or more letters, digits 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;
import package_name.*;
```

The package declaration indicates the package the sym and parser classes that are generated 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. As a part of the parser file, a separate non-public class is produced to contain all embedded user actions. The first action code declaration section allows code to be included in this class. Methods and fields for use by the code embedded in the grammar would normally be placed in this section (a typical example might be symbol table manipulation methods). This declaration takes the form:

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

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

After the action code declaration is an optional parser code declaration. This declaration allows methods and fields to be placed directly within the generated parser class. 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 initialise the lexical analyser as well as various tables and other data structures that might be needed by the 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 lexical analyser. 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.

Symbol Lists

Following user supplied code is the first required part of the specification: the symbol lists. These declarations are responsible for naming and supplying a type for each terminal and nonterminal symbol that appears in the grammar. As indicated above, each terminal and nonterminal symbol is represented at runtime by a Symbol object. In the case of terminals,

these are returned by the lexical analyser and placed on the parse stack. The lexical analyser should put the value of the terminal in the value field. Nonterminals replace a series of Symbol objects on the parse stack whenever the right hand side of some rule is reduced. In order to tell the parser which object types should be used for which symbol, terminal and nonterminal declarations are used. These take the forms:

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

The classname specified represents the type of the value of that terminal or nonterminal. When accessing these values through labels, the user uses the type declared. The classname can be of any type. If no classname is given, then the terminal or nonterminal holds no value. A label referring to such a symbol will have a null value.

Precedence and Associativity declarations

The third section, which is optional, specifies the precedences and associativity of terminals. More about this later.

The Grammar

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

```
start with nonterminal;
```

This indicates which nonterminal is the start or goal nonterminal for parsing. If a start nonterminal is not explicitly declared, then the nonterminal on the left hand side of the first rule 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 field contains the final reduction result.

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

Each symbol on the right hand side can optionally be labelled with a name. Label names appear after the symbol name separated by a colon (:). Label names must be unique within the rule, 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 nonterminal covers in the input file. These values must be properly initialised in the terminals by the lexical analyser. The left and right values then propagate to nonterminals to which rules reduce.

If there are several rules for the same nonterminal they may be declared together. In this case the rules start with the nonterminal and "::=". This is followed by multiple right hand sides each separated by a bar (|). The full set of rules 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 rule to the left of the action has been recognised. (Note that the lexical analyser will have returned the token one past the point of the action since the parser needs this extra lookahead token for recognition.)

Precedence specifications follow all the symbols and actions of the right hand side of the rule whose precedence it is assigning. Precedence specification allows a rule to be assigned a precedence not based on the last terminal in it. More about this later.

Command Line Options When Running CUP

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 "$LIB330/java_cup.jar" options < inputfile
```

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

In addition to the specification file, CUP's behaviour can also be changed by passing various options to it. Legal options 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 nonterminals 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 recognised right hand side of a rule with its left hand side). This is called a shift/reduce conflict. Similarly, the parser may not be able to decide between reductions with two different rules. 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 useful to resolve such conflicts. In this case CUP uses YACC convention and resolves shift/reduce conflicts by shifting, and reduce/reduce conflicts using the "highest priority" rule (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.

-nowarn

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

-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.

-dumpto file

This is a local improvement, to allow the diagnostic information to go to a file, rather than stderr. This is useful on PCs, that do not allow redirection of stderr. The file name is relative to the source directory.

-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.

-source sourceDir

This is a local improvement, to allow the directory for the source file to be specified in the options. If the –dest option is not specified, the source directory is also used as the destination directory for the generated files.

-dest destDir

This is a local improvement, to allow the directory for the generated files to be specified in the options.

-input fileName

This is a local improvement, to allow the CUP source file to be specified in the options, rather than by redirecting standard input. This is needed on systems other than UNIX, that do not support file redirection. The file name is relative to the source directory.

My typical command to run CUP in inside the createcompiler.bash shell script and is something like

```
java -jar "$CCUPJAR" -nonterms \
-expect 1 -progress -source "$SOURCEDIR" \
-dump -dumpto "parser.states" \
-input "parser.cup" &> cup.error
```

The structure of the Java Source Generated by CUP

Suppose we take our example INTERP1. What does the code generated by CUP look like?

The java_cup.runtime package must be used by any parser generated by CUP. CUP itself (apart from the first version) was written using CUP, so even CUP uses this package.

The lr_parser class in the java_cup.runtime package is an abstract class. It provides the general algorithm to implement any LALR(1) parser, in the parse() method. The algorithm corresponds roughly to

```
Start with a stack of the form
     state 0
     --->grows
where state 0 corresponds to having seen no input;

currToken = GetToken();

forever {
     /*
     Index the action table by
          the top of stack state
          and the current token
     to get the action to perform
     */
     CurrAction =  Action[ TopStack(), currToken ];
     switch ( CurrAction.ActionSort ) {
          case SHIFT:
               /*
               Push the indicated state onto the stack
               */
               currToken.setState( CurrAction.ShiftState );
               Push( currToken );
               /*
               Get the next token
               */
               currToken = GetToken();
               break;
          case REDUCE:
               /*
               Perform code associated with CurrAction.ReduceRule
                    (e.g., build node of tree corresponding to
                    the grammar rule);
               */
               nonTerminal = CurrAction.ReduceRule.do_action();
               /*
               Pop the states corresponding to
                    the righthand side of the rule
                    off the stack
               */
               Pop CurrentAction.ReduceRule.RHS.length()
                    states off the stack;
               /*
               Index the goto table by
                    the state uncovered on the top of stack
                    and the nonterminal corresponding to
                         the lefthand side of the rule
               and push the state found in the goto table
                    onto the stack;
               */
               nonTerminal.setState( GoTo[ TopStack(),
                    CurrentAction.ReduceRule.LHS ] );
               Push( nonTerminal );
               break;
          case ACCEPT:
               /*
               Complete parsing
               */
               return;
          case ERROR:
               /*
               Generate an error message
               */
```

```
                error();
                break;
            }
        }
```

The lr_parser class assumes that the rule, action, and reduce (goto) tables will be provided by the parser subclass generated by running CUP on the grammar we write. Similarly, it assumes that a method do_action() will be provided to correspond to the Java code in the actions for the grammar. There are a number of other abstract methods in this class, such as the scan() method, to obtain the next token.

The virtual_parse_stack class in the java_cup.runtime package provides additional support for parsing ahead in the event of a syntax error.

The Symbol class in the java_cup.runtime package represents the type of a terminal symbol returned by the lexical analyser and the terminal or nonterminal symbol pushed onto the stack by the parser. It contains fields to indicate the symbol kind, the state of the symbol, the value of the symbol, and the position of the left and right ends of the symbol in the input file.

The parser class created by running CUP extends the lr_parser class in the java_cup.runtime package. This class contains the appropriate tables (computed by CUP).

CUP copies the code from

```
parser code
     {:
     private Yylex lexer;
     private File file;

     public parser( File file ) {
          this();
          this.file = file;
          try {
               lexer = new Yylex( new FileReader( file ) );
               }
          catch ( IOException exception ) {
               throw new Error( "Unable to open file \"" + file + "\"" );
               }
          }
...
     :};
```

into the body of the parser class.

CUP copies the code from

```
scan with
     {:
     return  lexer.yylex();
     :};
```

into the scan() method within the parser class.

```
public java_cup.runtime.Symbol scan() throws java.lang.Exception {
     return  lexer.yylex();
     }
```

CUP creates a class CUP$parser$actions to encapsulate the user supplied action code, and creates an instance of this class as a field of the parser class.

CUP copies the code from

```
action code
     {:
     Hashtable table = new Hashtable();
     :};
```

into the body of the action class.

CUP also builds a method CUP$parser$do_action() within the action class, with a switch statement containing the user supplied action code. The references to the labels within the rules are replaced by references to the value field of the corresponding symbol on the stack. A new symbol is returned, with the symbol kind, the value of the symbol, and the position of the left and right ends of the symbol in the input file filled in.

```
  public final java_cup.runtime.Symbol CUP$parser$do_action(
     int                        CUP$parser$act_num,
     java_cup.runtime.lr_parser CUP$parser$parser,
     java.util.Stack            CUP$parser$stack,
     int                        CUP$parser$top)
     throws java.lang.Exception
     {
       /* Symbol object for return from actions */
       java_cup.runtime.Symbol CUP$parser$result;

       /* select the action based on the action number */
       switch (CUP$parser$act_num)
          {
            /*. . . . . .*/
            case 16: // Factor ::= IDENT
               {
                 Integer RESULT = null;
          int identleft =
((java_cup.runtime.Symbol)CUP$parser$stack.elementAt(CUP$parser$top-
0)).left;
          int identright =
((java_cup.runtime.Symbol)CUP$parser$stack.elementAt(CUP$parser$top-
0)).right;
          String ident = (String)((java_cup.runtime.Symbol)
CUP$parser$stack.elementAt(CUP$parser$top-0)).value;

          Integer value = ( Integer ) table.get( ident );
          if ( value == null ) {
               parser.report_error( "Undeclared Identifier " + ident,
               new Symbol( sym.IDENT, identleft, identright, ident ) );
               value = new Integer( 0 );
               }
          RESULT = value;

                 CUP$parser$result = new java_cup.runtime.Symbol(5/*Factor*/,
((java_cup.runtime.Symbol)CUP$parser$stack.elementAt(CUP$parser$top-
0)).left,
((java_cup.runtime.Symbol)CUP$parser$stack.elementAt(CUP$parser$top-
0)).right, RESULT);
               }
            return CUP$parser$result;
...
            /*. . . . . .*/
            case 14: // Factor ::= LEFT Expr RIGHT
               {
                 Integer RESULT = null;
          int exprleft =
((java_cup.runtime.Symbol)CUP$parser$stack.elementAt(CUP$parser$top-
1)).left;
          int exprright =
((java_cup.runtime.Symbol)CUP$parser$stack.elementAt(CUP$parser$top-
1)).right;
```

```
          Integer expr = (Integer)((java_cup.runtime.Symbol)
CUP$parser$stack.elementAt(CUP$parser$top-1)).value;

          RESULT = expr;

                CUP$parser$result = new java_cup.runtime.Symbol(5/*Factor*/,
((java_cup.runtime.Symbol)CUP$parser$stack.elementAt(CUP$parser$top-
2)).left,
((java_cup.runtime.Symbol)CUP$parser$stack.elementAt(CUP$parser$top-
0)).right, RESULT);
              }
            return CUP$parser$result;
...
```

CUP also generates the file sym.java. This file declares symbolic constants corresponding to each kind of symbol. My modified version of CUP also generates text corresponding to the names of symbols, and the text for the grammar rules. This information is used when performing a debug_parse(), to produce better diagnostic messages.

```
public class sym {
     /* terminals */
     public static final int MINUS = 6;
     public static final int IDENT = 11;
     public static final int DIVIDE = 8;
     public static final int LEFT = 2;
     public static final int NUMBER = 10;
...

     /* nonterminals */
     static final int Factor = 5;
     static final int Term = 4;
     static final int Stmt = 2;
     static final int Expr = 3;
     static final int $START = 0;
     static final int StmtList = 1;
     public static String terminal_name( int id ) {
          switch ( id ) {
               case 6: return "MINUS";
               case 11: return "IDENT";
               case 8: return "DIVIDE";
               case 2: return "LEFT";
               case 10: return "NUMBER";
               case 0: return "EOF";
...
               }
          }

     public static String non_terminal_name( int id ) {
          switch ( id ) {
               case 5: return "Factor";
               case 4: return "Term";
               case 2: return "Stmt";
               case 3: return "Expr";
               case 0: return "$START";
               case 1: return "StmtList";
               default: return "unknown non_terminal" + id;
               }
          }
     public static String rule_name( int id ) {
          switch ( id ) {
               case 16: return "Factor ::= IDENT ";
               case 15: return "Factor ::= NUMBER ";
```

```
                case 14: return "Factor ::= LEFT Expr RIGHT ";
                case 13: return "Term ::= Factor ";
                case 12: return "Term ::= Term DIVIDE Factor ";
                case 11: return "Term ::= Term TIMES Factor ";
                case 10: return "Expr ::= Term ";
...
                }
          }
}
```

Finally, CUP can generate a summary of the states, and action and goto tables.

```
===== Terminals =====
[0] EOF [1] error [2] LEFT [3] RIGHT [4] NEWLINE
[5] PLUS [6] MINUS [7] TIMES [8] DIVIDE [9] ASSIGN
[10] NUMBER [11] IDENT

===== Nonterminals =====
[0] $START [1] StmtList [2] Stmt [3] Expr [4] Term
[5] Factor

===== Rules =====
[0] $START ::= StmtList EOF
[1] StmtList ::=
[2] StmtList ::= StmtList Stmt
[3] Stmt ::= IDENT ASSIGN Expr NEWLINE
[4] Stmt ::= Expr NEWLINE
[5] Stmt ::= error NEWLINE
[6] Stmt ::= NEWLINE
[7] Expr ::= Expr PLUS Term
[8] Expr ::= Expr MINUS Term
[9] Expr ::= MINUS Term
[10] Expr ::= Term
[11] Term ::= Term TIMES Factor
[12] Term ::= Term DIVIDE Factor
[13] Term ::= Factor
[14] Factor ::= LEFT Expr RIGHT
[15] Factor ::= NUMBER
[16] Factor ::= IDENT

===== LALR(1) States =====
START lalr_state [0]: {
  [StmtList ::= (*) StmtList Stmt , {EOF error LEFT NEWLINE MINUS NUMBER
IDENT }]
  [StmtList ::= (*) , {EOF error LEFT NEWLINE MINUS NUMBER IDENT }]
  [$START ::= (*) StmtList EOF , {EOF }]
}
transition on StmtList to state [1]

-------------------
lalr_state [1]: {
  [Factor ::= (*) LEFT Expr RIGHT , {NEWLINE PLUS MINUS TIMES DIVIDE }]
  [Term ::= (*) Term TIMES Factor , {NEWLINE PLUS MINUS TIMES DIVIDE }]
  [Expr ::= (*) Expr MINUS Term , {NEWLINE PLUS MINUS }]
  [StmtList ::= StmtList (*) Stmt , {EOF error LEFT NEWLINE MINUS NUMBER
IDENT }]
  [Stmt ::= (*) error NEWLINE , {EOF error LEFT NEWLINE MINUS NUMBER IDENT
}]
  [Factor ::= (*) IDENT , {NEWLINE PLUS MINUS TIMES DIVIDE }]
  [Term ::= (*) Factor , {NEWLINE PLUS MINUS TIMES DIVIDE }]
  [Expr ::= (*) Term , {NEWLINE PLUS MINUS }]
  [Expr ::= (*) Expr PLUS Term , {NEWLINE PLUS MINUS }]
```

```
  [Stmt ::= (*) Expr NEWLINE , {EOF error LEFT NEWLINE MINUS NUMBER IDENT }]
  [Factor ::= (*) NUMBER , {NEWLINE PLUS MINUS TIMES DIVIDE }]
  [Term ::= (*) Term DIVIDE Factor , {NEWLINE PLUS MINUS TIMES DIVIDE }]
  [Expr ::= (*) MINUS Term , {NEWLINE PLUS MINUS }]
  [Stmt ::= (*) NEWLINE , {EOF error LEFT NEWLINE MINUS NUMBER IDENT }]
  [$START ::= StmtList (*) EOF , {EOF }]
  [Stmt ::= (*) IDENT ASSIGN Expr NEWLINE , {EOF error LEFT NEWLINE MINUS
NUMBER IDENT }]
}
transition on NEWLINE to state [12]
transition on Factor to state [11]
transition on error to state [10]
transition on Term to state [9]
transition on Stmt to state [8]
transition on EOF to state [7]
transition on NUMBER to state [6]
transition on LEFT to state [5]
transition on Expr to state [4]
transition on IDENT to state [3]
transition on MINUS to state [2]
...
-------- ACTION_TABLE --------
From state #0
     EOF:REDUCE(rule 1) error:REDUCE(rule 1) LEFT:REDUCE(rule 1)
     NEWLINE:REDUCE(rule 1) MINUS:REDUCE(rule 1) NUMBER:REDUCE(rule 1)
     IDENT:REDUCE(rule 1)
From state #1
     EOF:SHIFT(state 7) error:SHIFT(state 10) LEFT:SHIFT(state 5)
     NEWLINE:SHIFT(state 12) MINUS:SHIFT(state 2) NUMBER:SHIFT(state 6)
     IDENT:SHIFT(state 3)
From state #2
     LEFT:SHIFT(state 5) NUMBER:SHIFT(state 6) IDENT:SHIFT(state 17)
From state #3
     NEWLINE:REDUCE(rule 16) PLUS:REDUCE(rule 16) MINUS:REDUCE(rule 16)
     TIMES:REDUCE(rule 16) DIVIDE:REDUCE(rule 16) ASSIGN:SHIFT(state 26)
...
-------- REDUCE_TABLE --------
From state #0:
     StmtList:GOTO(1)
From state #1:
     Stmt:GOTO(8)
     Expr:GOTO(4)
     Term:GOTO(9)
     Factor:GOTO(11)
From state #2:
     Term:GOTO(29)
     Factor:GOTO(11)
From state #3:
...
```

What they call productions, I call rules. What they call the reduce table, I call the goto table.

The states are very useful when debugging a grammar that has conflicts in it.