Chapter 5 Using Ambiguous Grammars

It is often convenient to write simple ambiguous grammars for languages, rather than writing a full unambiguous grammar. For example, many computer languages have two forms of if statement:

```
Stmt::=
          IF LEFT Expr RIGHT Stmt
     |
          IF LEFT Expr RIGHT Stmt ELSE Stmt
     |
          OtherStmts
     ;
```

This grammar is in fact ambiguous. If we nest two if statements, the grammar does not tell us which if statement the else part matches with. In other words, do we parse

```
if ( C1 ) if ( C2 ) S1 else S2
```

as

```
if ( C1 ) {
     if ( C2 )
          S1
     else
          S2
     }
```

or

```
if ( C1 ) {
     if ( C2 )
          S1
     }
else
     S2
```

We all know that it is the first interpretation that is given - the “ELSE” is matched with the most recent if statement. Indeed, any left to right shift-reduce parse would have to do it this way, since there is no way of telling, at the point at which the “ELSE” is met, whether a second “ELSE” will follow.

If we generate LALR(1) parsing tables for the above grammar, we find that there is a shift/reduce conflict, when we have “IF LEFT Expr RIGHT IF LEFT Expr RIGHT Stmt” on the stack, and the current token is “ELSE”. Deciding to match the “ELSE” to the most recent if statement amounts to performing a shift, rather than a reduction.

Attempting to write an unambiguous grammar to eliminate the “dangling else” problem is not easy, because not only do we get problems with directly nested if statements, but also with constructs such as

```
if ( C1 )
     while ( C2 )
          if ( C3 )
               S1
          else
               S2
```

We need to duplicate the grammar rules for all control structures that lack a clear marker for the end of the statement, namely labelled, if, while, and for statements. For example in Java (refer JAVA)

```
Statement::=
          StatementWithoutTrailingSubstatement
```

```
     |
          LabeledStatement
     |
          IfThenStatement
     |
          IfThenElseStatement
     |
          WhileStatement
     |
          ForStatement
     ;

StatementNoShortIf::=
          StatementWithoutTrailingSubstatement
     |
          LabeledStatementNoShortIf
     |
          IfThenElseStatementNoShortIf
     |
          WhileStatementNoShortIf
     |
          ForStatementNoShortIf
     ;

StatementWithoutTrailingSubstatement::=
          Block
     |
          EmptyStatement
     |
          ExpressionStatement
     |
          SwitchStatement
     |
          DoStatement
     |
          BreakStatement
     |
          ContinueStatement
     |
          ReturnStatement
     |
          SynchronizedStatement
     |
          ThrowStatement
     |
          TryStatement
     ;

LabeledStatement::=
          IDENTIFIER ":" Statement
     ;

LabeledStatementNoShortIf::=
          IDENTIFIER ":" StatementNoShortIf
     ;

IfThenStatement::=
          "if" "(" Expression ")" Statement
     ;

IfThenElseStatement::=
```

```
          "if" "(" Expression ")" StatementNoShortIf "else" Statement
     ;

IfThenElseStatementNoShortIf::=
          "if" "(" Expression ")"
          StatementNoShortIf "else" StatementNoShortIf
     ;

WhileStatement::=
          "while" "(" Expression ")" Statement
     ;

WhileStatementNoShortIf::=
          "while" "(" Expression ")" StatementNoShortIf
     ;

ForStatement::=
          "for"  "("  ForInit_Opt  ";"  Expression_Opt  ";"  ForUpdate_Opt  ")"
Statement
     ;

ForStatementNoShortIf::=
          "for" "(" ForInit_Opt ";" Expression_Opt ";" ForUpdate_Opt ")"
               StatementNoShortIf
     ;
```

Clearly, it is painful trying to specify that the "then" part of an "if-then-else" statement must have matching "then"s and "else"s.

Using Ambiguous Grammars and Precedences

Another situation in which ambiguous grammars are useful is in grammars for expressions.

Operators typically have a precedence and associativity, which is used to resolve ambiguity when parsing expressions involving operators.

Operators are ranked according to their precedence. For example "*" and "/" have a higher precedence than "+" and "-". We associate an operand between two operators of different precedence with the higher precedence operator. For example "a + b * c" is interpreted as "a + (b * c)", and "a * b + c" is interpreted as "(a * b) + c".

If two operators have the same precedence, we use what is called associativity to resolve the ambiguity. Operators can be either left associative, right associative, or nonassociative. If two operators are of the same precedence, we associate an operand between these operators with the left operator, if the operators are left associative, or the right operand, if the operators are right associative, and we don't permit such situations, if the operators are non-associative. All infix operators in Java and mathematics are left associative, apart from the assignment operators. Since "+" and "-" are left associative "a - b + c" is interpreted as (a - b) + c". Since "=" is right associative, "a = b = c" is interpreted as "a = (b = c)". In some languages "<" is nonassociative, so "a < b < c" is illegal.

For example, in Java, the operator precedences are roughly

Prec	Assoc	Operands	Operator
16	n/a	0	Names, literals, parenthesised expressions.
15	left	2	Subscripting, method invocations, field selection, new.
14	left	1	Postfix operators.
13	right	1	Prefix operators, casts.
12	left	2	Multiplicative operators.
11	left	2	Additive operators
10	left	2	Shift operators.
9	left	2	Relational, instanceof operators.
8	left	2	Equality operators.
7	left	2	Bitwise and.
6	left	2	Bitwise exlusive or.
5	left	2	Bitwise inclusive or.
4	left	2	Logical and.
3	left	2	Logical or.
2	right	3	Conditional operator
1	right	2	Assignment operators.

(Casts don't fit into the pattern perfectly, since their operands can't start with a "+" or "-", and "new" and "...?...:..." are both a bit strange.)

We can specify the precedences and associativity of operators, as a part of the grammar. However, it is painful having to specify that for left associative, precedence n operators

```
Prec_nExpr::=
        Prec_nExpr Prec_nOpr_1 Prec_n+1Expr
    |
        Prec_nExpr Prec_nOpr_2 Prec_n+1Expr
    |
        Prec_nExpr Prec_nOpr_3 Prec_n+1Expr
    |
        Prec_n+1Expr
    ;
```

particularly if we have large numbers of operators, of many different precedences, as is the case in Java and C.

It is sometimes easier to use an ambiguous grammar, for which we omit the information about operator precedence and associativity from the grammar. We can then specify the precedence/associativity of operators in separate declarations. Nevertheless, defining precedences and associativity to resolve ambiguities is less flexible than using a full grammar, and while a little verbosity is added to the grammar definition, it is probably safer to write an unambiguous grammar without using precedences. Specifying precedences without due care can cause major design flaws in the grammar to be hidden.

```
precedence right ASSIGN;
precedence left PLUS, MINUS;
precedence left STAR, DIVIDE;
precedence right PREFIX;
precedence left INCR, DECR;

Expr::=
          Expr ASSIGN Expr
     |
          Expr PLUS Expr
     |
          Expr MINUS Expr
     |
          Expr STAR Expr
     |
          Expr DIVIDE Expr
     |
          MINUS Expr
          %prec PREFIX
     |
          INCR Expr
          %prec PREFIX
     |
          DECR Expr
          %prec PREFIX
     |
          STAR Expr
          %prec PREFIX
     |
          AMPERSAND Expr
          %prec PREFIX
     |
          Expr INCR
     |
          Expr DECR
     |
          Ident LEFT ExprListOpt RIGHT
     |
          LEFT Expr RIGHT
     |
          NUMBER
     |
          Ident
     ;

ExprListOpt::=
          /* Empty */
     |
          ExprList
     ;
```

Conflict resolution in CUP

How are precedences used in shift-reduce parsing? We declare a precedence and associativity for our terminal symbols. (We specify precedence and associativity together, so we can't have different terminal symbols of the same precedence but different associativity.)

A precedence is required for terminal symbols used in left recursive rules of the form "Expr → Expr OPR2 β", such as infix and postfix operators. "β" is often empty (postfix operator) or "Expr" (infix operator).

A precedence is required for right recursive rules of the form "Expr → α OPR1 Expr", such as infix and prefix operators. "α" is often "Expr" (infix operator) or empty (prefix operator).

The precedence for a rule is normally the precedence of the last terminal symbol on the right hand side ("OPR1"). However, it is also possible to explicitly specify a precedence for a rule. This might need to be done when we use the same operator as both an infix and a prefix operator (for example "-") or both a prefix and a postfix operator (for example "++"), if the prefix operator has a different precedence from the infix or postfix operator. We declare a nonexistent terminal symbol such as "PREFIX", and give it the precedence and associativity of the prefix operator. We append "%prec PREFIX" to the right hand side of the rule involving its use as a prefix operator to specify that it has the same precedence as "PREFIX".

When we have shift/reduce conflicts, the precedence and associativity of the rule on the top of stack, and the current token are used to determine whether to shift or reduce. CUP does the following.

Shift/Reduce conflicts

Rule Precedence	>	Token Precedence	=>	Reduce
Rule Precedence	<	Token Precedence	=>	Shift
Rule Precedence	==	Token Precedence		
		Left Associative	=>	Reduce
		Right Associative	=>	Shift
		NonAssociative	=>	Produce error entry in parsing table
No Relation			=>	Shift and report as Shift/Reduce conflict.

Reduce/Reduce conflicts

=>	Reduce by earlier grammar rule in grammar definition, and report as Reduce/Reduce conflict.

Reduce/reduce conflicts almost always represent fundamental design flaws in the grammar. Shift/reduce conflicts may indicate a need to define a precedence for additional operators, but should definitely not be ignored, since the parser might be resolving the conflict in the opposite way from the intended way. Apart from the dangling else problem, shift/reduce conflicts involving terminals other than operators are almost certainly fundamental design flaws in the grammar.

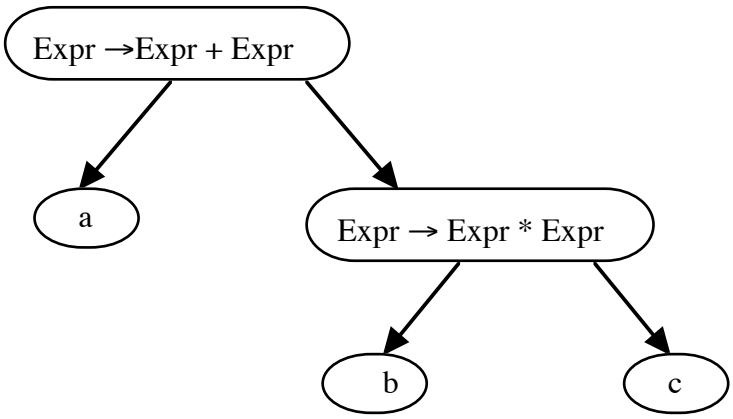
For example, suppose we have rules "Expr → Expr + Expr" and "Expr → Expr * Expr", with "*" having a higher precedence than "+".

Suppose we have the input "a + b * c". this generates a parse:

Stack	**Current Token**	**Action**
$	a	Shift Ident a
$ Ident	+	Reduce Expr → Ident
$ Expr		Shift +
$ Expr +	b	Shift Ident b
$ Expr + Ident	*	Reduce Expr → Ident
$ Expr + Expr		**Shift *, because * has higher prec**
$ Expr + Expr *	c	Shift Ident c
$ Expr + Expr * Ident	$	Reduce Expr → Ident
$ Expr + Expr * Expr		Reduce Expr → Expr * Expr

$ Expr + Expr	Reduce Expr → Expr + expr
$ Expr	Accept

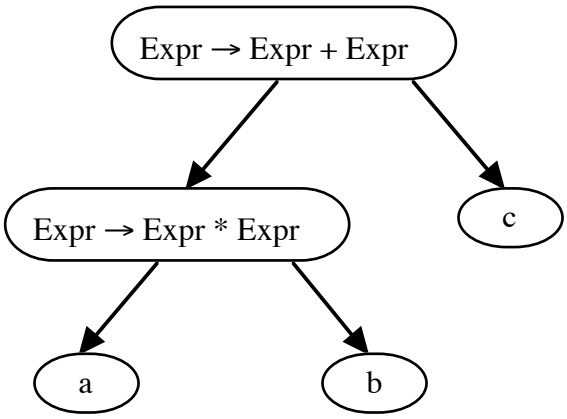
and generates a tree



Suppose we have the input "a * b + c". this generates a parse:

Stack	**Current Token**	**Action**
$	a	Shift Ident a
$ Ident	*	Reduce Expr → Ident
$ Expr		Shift *
$ Expr *	b	Shift Ident b
$ Expr * Ident	+	Reduce Expr → Ident
$ Expr * Expr		**Reduce Expr → Expr * Expr, because * has higher prec**
$ Expr		Shift +
$ Expr +	c	Shift Ident c
$ Expr + Ident	$	Reduce Expr → Ident
$ Expr + Expr		Reduce Expr → Expr + expr
$ Expr		Accept

and generates a tree



Assigning precedences to terminals and rules

Suppose we use a single nonteminal "Expr" for all expressions, and use precedences to resolve conflicts.

Suppose that we have both left and right recursive rules involving Expr. For example, we might have a right recursive rule of the form "Expr → α OPR1 Expr" and a left recursive rule

of the form "Expr → Expr OPR2 β". They might even be the same rule for an infix operator "Expr → Expr OPR Expr".

Suppose we have input "α OPR1 Expr OPR2 β". There are two ways of matching this input to Expr.

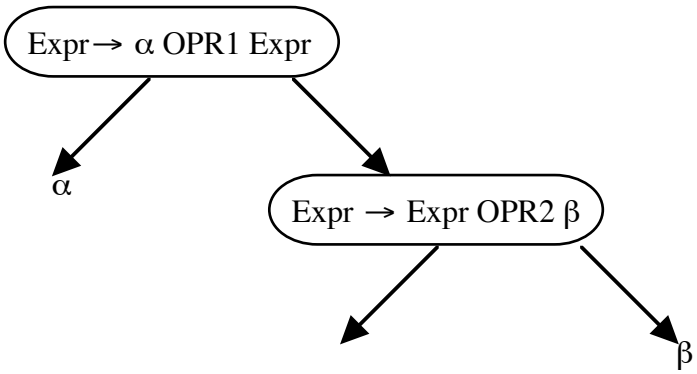
When we perform a bottom up parse, the following situation will develop.

Stack	**Remaining input**	**Action**
... α OPR1 Expr	OPR2 β ...	

We can either shift first, then reduce

Stack	**Remaining input**	**Action**
... α OPR1 Expr	OPR2 β ...	Shift OPR2 β
... α OPR1 Expr OPR2 β	...	Reduce Expr → Expr OPR2 β
... α OPR1 Expr	...	Reduce Expr → α OPR1 Expr
... Expr	...	

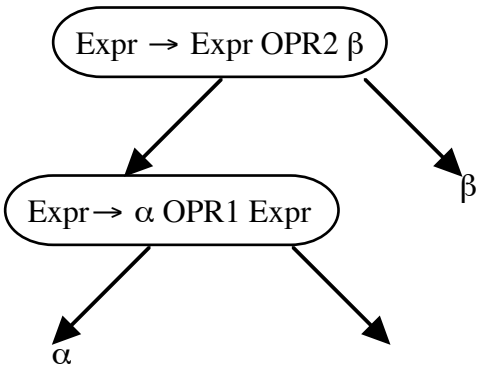
and generate a tree



or reduce first, then shift

Stack	**Remaining input**	**Action**
... α OPR1 Expr	OPR2 β ...	Reduce Expr → α OPR1 Expr
... Expr	OPR2 β ...	Shift OPR2 β
... Expr OPR2 β	...	Reduce Expr → Expr OPR2 β
... Expr	...	

and generate a tree



The grammar is clearly ambiguous, so no parser can avoid having a conflict. However it can resolve the conflict by choosing whether to shift or reduce, based on precedences and associativity.

We need a precedence for the rule "Expr → α OPR1 Expr" (normally the precedence of OPR1), and the operator OPR2. The precedence relationship is used to determine whether to shift or reduce when "α OPR1 Expr" is on the stack, and the current token is OPR2.

If OPR1 and OPR2 are the same precedence, we resolve the conflict by taking into account associativity. For example, we could have a rule that is both left and right recursive, as in "Expr → Expr OPR Expr" (so OPR is an infix operator).

If an operator is used in both left and right recursive rules then the precedence of the right recursive rule sometimes needs to be specified by a "%prec" specification. This happens for operators that are used as both prefix and infix operators, or both prefix and postfix operators, with different precedences.

The specification of a precedence applies not only for terminals and rules that we think of as involving operators, but also rules such as "Expr → Expr LEFT ExprSeq RIGHT" (function invocations), and "Expr → Expr QUEST Expr COLON Expr" (conditional expressions).

The normal interpretation in C and Java can be provided by giving LEFT a high precedence, and QUEST and COLON a low (right associative) precedence.

The precedence is used when we have something like "a + f(x, y)" or "a < b ? c : d < e ? f : g".

Both of these are a little strange, in that the ExprSeq inside "LEFT ExprSeq RIGHT" and the Expr inside "QUEST Expr COLON" have a very low precedence, but this system still works, because once LEFT or QUEST has been shifted, there are no conflicts to be resolved. This illustrates that precedences are somewhat "Mickey Mouse" (lacking in any coherent logic), and a purist would tell you not to use them. There is no guarantee that the operands of an operator have a higher or equal precedence to the operator.

Generating Abstract Syntax Trees and Using Precedences (Refer REPRINT)

Lets create a parser that instead of evaluating expressions, builds an abstract syntax tree. An abstract syntax tree is a tree representing the structure of the input, without the syntactic sugar of parentheses, braces, etc, that are used to enclose low precedence expressions to make them into high precedence expressions, or enclose statement sequences to make them into a single statement. We need to do something with the tree once we have it, such as print the tree out again.

The lexical analyser

The lexical analyser is not very different from before, except I have added some operators, namely "++", "--" and "&".

```
package grammar;

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

%%

%public
%char
%type          Symbol

%{
     private int lineNumber = 1;
```

```
     public int lineNumber() { return lineNumber; }

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

%}

%init{
     yybegin( NORMAL );
%init}

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

%state NORMAL LEXERROR

%%

<NORMAL> {
     "="            { return token( sym.ASSIGN ); }
     "+"            { return token( sym.PLUS ); }
     "-"            { return token( sym.MINUS ); }
     "*"            { return token( sym.STAR ); }
     "/"            { return token( sym.DIVIDE ); }
     "++"           { return token( sym.INCR ); }
     "--"           { return token( sym.DECR ); }
     "&"            { return token( sym.AMPERSAND ); }
     ","            { return token( sym.COMMA ); }
     "("            { return token( sym.LEFT ); }
     ")"            { return token( sym.RIGHT ); }
     {newline}      { lineNumber++; return token( sym.NEWLINE ); }
     {space}        { }

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

     .              {
                    yybegin( LEXERROR );
                    return token( sym.error );
                    }
     }

<LEXERROR> {
     {newline}      {
                    lineNumber++;
                    yybegin( NORMAL );
                    return token( sym.NEWLINE );
                    }
     .              { }
     }

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

The parser

This time, the parser builds an object representing an abstract syntax tree, rather than attempting to evaluate the expressions. The other big difference is that I use an ambiguous grammar, and resolve conflicts by precedences.

```
package grammar;

import node.*;
import node.stmtNode.*;
import node.exprNode.*;
import node.exprNode.prefixNode.*;
import node.exprNode.postfixNode.*;
import node.exprNode.binaryNode.*;

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

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, STAR, DIVIDE, ASSIGN,
     INCR, DECR, AMPERSAND, COMMA, PREFIX;
terminal String NUMBER;
terminal String IDENT;

nonterminal StmtListNode StmtList;
nonterminal StmtNode Stmt;
nonterminal ExprNode Expr;
nonterminal ExprListNode ExprListOpt, ExprList;
nonterminal IdentNode Ident;

precedence right ASSIGN;
precedence left PLUS, MINUS;
precedence left STAR, DIVIDE;
precedence right PREFIX;
precedence left INCR, DECR;
```

```
start with StmtList;

StmtList::=
          {:
          RESULT = new StmtListNode();
          :}
     |
          StmtList:stmtList Stmt:stmt
          {:
          stmtList.addElement( stmt );
          RESULT = stmtList;
          :}
     ;

Stmt::=
          Expr:expr NEWLINE
          {:
          RESULT = new PrintStmtNode( expr );
          :}
     |
          error NEWLINE
          {:
          RESULT = new ErrorStmtNode();
          :}
     |
          NEWLINE
          {:
          RESULT = new NullStmtNode();
          :}
     ;

Expr::=
          Expr:expr1 ASSIGN Expr:expr2
          {:
          RESULT = new AssignNode( expr1, expr2 );
          :}
     |
          Expr:expr1 PLUS Expr:expr2
          {:
          RESULT = new PlusNode( expr1, expr2 );
          :}
     |
          Expr:expr1 MINUS Expr:expr2
          {:
          RESULT = new MinusNode( expr1, expr2 );
          :}
     |
          Expr:expr1 STAR Expr:expr2
          {:
          RESULT = new TimesNode( expr1, expr2 );
          :}
     |
          Expr:expr1 DIVIDE Expr:expr2
          {:
          RESULT = new DivideNode( expr1, expr2 );
          :}
     |
          MINUS Expr:expr
          {:
          RESULT = new NegateNode( expr );
          :}
```

```
          %prec PREFIX
     |
          INCR Expr:expr
          {:
          RESULT = new PreIncrNode( expr );
          :}
          %prec PREFIX
     |
          DECR Expr:expr
          {:
          RESULT = new PreDecrNode( expr );
          :}
          %prec PREFIX
     |
          STAR Expr:expr
          {:
          RESULT = new ContentsNode( expr );
          :}
          %prec PREFIX
     |
          AMPERSAND Expr:expr
          {:
          RESULT = new AddressNode( expr );
          :}
          %prec PREFIX
     |
          Expr:expr INCR
          {:
          RESULT = new PostIncrNode( expr );
          :}
     |
          Expr:expr DECR
          {:
          RESULT = new PostDecrNode( expr );
          :}
     |
          Ident:proc LEFT ExprListOpt:exprListOpt RIGHT
          {:
          RESULT = new ProcInvocNode( proc, exprListOpt );
          :}
     |
          LEFT Expr:expr RIGHT
          {:
          RESULT = expr;
          :}
     |
          NUMBER:value
          {:
          RESULT = new NumberNode( new Integer( value ) );
          :}
     |
          Ident:expr
          {:
          RESULT = expr;
          :}
     ;

ExprListOpt::=
          /* Empty */
          {:
          RESULT = new ExprListNode();
```

```
          :}
     |
          ExprList:exprList
          {:
          RESULT = exprList;
          :}
     ;

ExprList::=
          Expr:expr
          {:
          ExprListNode exprList = new ExprListNode();
          exprList.addElement( expr );
          RESULT = exprList;
          :}
     |
          ExprList:exprList COMMA Expr:expr
          {:
          exprList.addElement( expr );
          RESULT = exprList;
          :}
     ;

Ident::=
          IDENT:ident
          {:
          RESULT = new IdentNode( ident );
          :}
     ;
```

The terminal declarations include a pseudoterminal, namely PREFIX, that is never returned by the lexical analyser.

```
terminal
     LEFT, RIGHT, NEWLINE, PLUS,
     MINUS, STAR, DIVIDE, ASSIGN,
     INCR, DECR, AMPERSAND, COMMA, PREFIX;
terminal String NUMBER;
terminal String IDENT;
```

Nonterminals have a type corresponding to a node of an abstract syntax tree.

```
nonterminal StmtListNode StmtList;
nonterminal StmtNode Stmt;
nonterminal ExprNode Expr;
nonterminal ExprListNode ExprListOpt, ExprList;
nonterminal IdentNode Ident;
```

The grammar I have written is actually ambiguous (there is more than one way of parsing input, such as “a + b * c”). I declare a precedence for my operators, and use the precedence to resolve ambiguities. Terminals in the same precedence declaration have the same precedence. Earlier declarations have a lower precedence than later declarations.

```
precedence right ASSIGN;
precedence left PLUS, MINUS;
precedence left STAR, DIVIDE;
precedence right PREFIX;
precedence left INCR, DECR;
```

Rules using an operator as a prefix operator need a “%prec” specification.

```
Expr::=
          ...
     |
          Expr:expr1 MINUS Expr:expr2
```

```
          {:
          RESULT = new MinusNode( expr1, expr2 );
          :}
     |
          ...
     |
          MINUS Expr:expr
          {:
          RESULT = new NegateNode( expr );
          :}
          %prec PREFIX
     |
          INCR Expr:expr
          {:
          RESULT = new PreIncrNode( expr );
          :}
          %prec PREFIX
     |
          ...
     ;
```

Note the action for parentheszed expressions

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

The abstract syntax tree does not contain a node to represent the parenthesization. Once we know the structure of the expression, we no longer need the parentheses.

The Main class

Again, we have a Main class to create the lexical analyser and parser, and initiate parsing.

```
import node.*;
import node.stmtNode.*;
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" ) );
            StmtListNode stmtList = ( StmtListNode ) p.parse().value;
            //   StmtListNode stmtList =
            //        ( StmtListNode ) p.debug_parse().value;
            System.out.println( "Reprinting ..." );
            System.out.print( stmtList );
            }
        catch ( Exception e ) {
            System.out.println( "Exception" );
            }
        }
    }}
```

The Node classes

I need to create classes to represent nodes of the abstract syntax tree. All nodes have a toString() method.

```
public abstract class Node {
    public abstract String toString();
    }
```

All nodes for expressions have an operator precedence. If the precedence of the surrounding context is greater than that of the node, it should be displayed enclosed in parentheses.

```
public abstract class ExprNode extends Node {

    protected final static int PREC_ASSIGN    =    0;
    protected final static int PREC_ADD       =    1;
    protected final static int PREC_MUL       =    2;
    protected final static int PREC_PREFIX    =    3;
    protected final static int PREC_POSTFIX   =    4;
    protected final static int PREC_PRIMARY   =    5;

    protected int precedence;

    public String toString( int parentPrec ) {
        if ( precedence < parentPrec )
            return "( " + toString() + " )";
        else
            return toString();
        }
    }
```

Assignment operators have two children, and are right associative. This means the left child has higher precedence, and the right child has the same precedence.

```
public class AssignNode extends ExprNode {

    String operator;
    ExprNode left, right;

    public AssignNode( ExprNode left, ExprNode right ) {
        this.left = left;
        this.right = right;
```

```
          precedence = PREC_ASSIGN;
          operator = "=";
          }

     public String toString() {
          return
               left.toString( precedence + 1 )
               + " " +  operator + " "
               + right.toString( precedence );
          }
     }
```

Infix operators are left associative. The left operand has the same precedence, and the right operand has a higher precedence.

```
public class BinaryNode extends ExprNode {

     String operator;
     ExprNode left, right;

     public BinaryNode( ExprNode left, ExprNode right ) {
          this.left = left;
          this.right = right;
          }

     public String toString() {
          return
               left.toString( precedence )
               + " " +  operator + " "
               + right.toString( precedence + 1 );
          }
     }
```

Addition and subtraction are left associative infix operators.

```
public class PlusNode extends BinaryNode {

     public PlusNode( ExprNode left, ExprNode right ) {
          super( left, right );
          precedence = PREC_ADD;
          operator = "+";
          }
     }

public class MinusNode extends BinaryNode {

     public MinusNode( ExprNode left, ExprNode right ) {
          super( left, right );
          precedence = PREC_ADD;
          operator = "-";
          }
     }
```

Multiplication and division are left associative infix operators.

```
public class TimesNode extends BinaryNode {

     public TimesNode( ExprNode left, ExprNode right ) {
          super( left, right );
          precedence = PREC_MUL;
          operator = "*";
          }
```

```
     }

public class DivideNode extends BinaryNode {

     public DivideNode( ExprNode left, ExprNode right ) {
          super( left, right );
          precedence = PREC_MUL;
          operator = "/";
          }
     }
```

Prefix operators have a right operand of the same precedence.

```
public class PrefixNode extends ExprNode {

     String operator;
     ExprNode right;

     public PrefixNode( ExprNode right ) {
          this.right = right;
          precedence = PREC_PREFIX;
          }

     public String toString() {
          return
               operator + " "
               + right.toString( precedence );
          }
     }

public class NegateNode extends PrefixNode {

     public NegateNode( ExprNode right ) {
          super( right );
          operator = "-";
          }
     }

public class PreIncrNode extends PrefixNode {

     public PreIncrNode( ExprNode right ) {
          super( right );
          operator = "++";
          }
     }

public class PreDecrNode extends PrefixNode {

     public PreDecrNode( ExprNode right ) {
          super( right );
          operator = "--";
          }
     }

public class AddressNode extends PrefixNode {

     public AddressNode( ExprNode right ) {
          super( right );
          operator = "&";
          }
     }
```

```
public class ContentsNode extends PrefixNode {

     public ContentsNode( ExprNode right ) {
          super( right );
          operator = "*";
          }
     }
```

Postfix operators have a left operand of the same precedence.

```
public class PostfixNode extends ExprNode {

     String operator;
     ExprNode left;

     public PostfixNode( ExprNode left ) {
          this.left = left;
          precedence = PREC_POSTFIX;
          }

     public String toString() {
          return
               left.toString( precedence )
               + " " +  operator;
          }
     }

public class PostIncrNode extends PostfixNode {

     public PostIncrNode( ExprNode left ) {
          super( left );
          operator = "++";
          }
     }

public class PostDecrNode extends PostfixNode {

     public PostDecrNode( ExprNode right ) {
          super( right );
          operator = "--";
          }
     }
```

Procedure invocations have a high precedence.

```
public class ProcInvocNode extends ExprNode {

     IdentNode ident;
     ExprListNode params;

     public ProcInvocNode( IdentNode ident, ExprListNode params ) {
          precedence = PREC_PRIMARY;
          this.ident = ident;
          this.params = params;
          }

     public String toString() {
          if ( params.size() == 0 )
               return ident.toString() + "()";
          else
               return ident.toString() + "( " + params.toString() + " )";
          }
```

```
     }

public class ExprListNode {

     private Vector list = new Vector();

     public ExprListNode() {
          }

     public void addElement( ExprNode node ) {
          list.addElement( node );
          }

     public int size() {
          return list.size();
          }

     public String toString() {
          String result = "";
          for ( int i = 0; i < list.size(); i++ ) {
               if ( i > 0 )
                    result += ", ";
               ExprNode expr = ( ExprNode ) list.elementAt( i );
               result += expr;
               }
          return result;
          }

     }
```

Numbers and identifiers have a high precedence.

```
public class NumberNode extends ExprNode {
     Integer value;
     public NumberNode( Integer value ) {
          this.value = value;
          precedence = PREC_PRIMARY;
          }

     public String toString() {
          return value.toString();
          }
     }

public class IdentNode extends ExprNode {
     String name;
     public IdentNode( String name ) {
          this.name = name;
          precedence = PREC_PRIMARY;
          }

     public String toString() {
          return name;
          }
     }
```

The above gives a very object oriented way of writing a compiler. It is a little verbose, having to write all these class declarations, and put each one in a separate file. We create a class for every kind of nonterminal, then extend this class for every rule with that nonterminal as its left hand side.

Every nonterminal node has fields that represent "attributes" of the construct, or methods to evaluate the "attributes" of the construct. For example, an expression node could be expected to have a field or method representing the type of the construct, the text to represent the expression, the code to evaluate the expression, etc. In some cases, the attributes could be evaluated and stored in the node. In other cases, a method could be provided to evaluate the attribute. For example, it is reasonable to have a method that takes a symbol table as a parameter, and returns the type of the expression.

The nodes in the tree represent the rules applied to match the input. They contain fields to point to the children, and specify other attributes, such as the value of a terminal node representing a constant, or the text of a terminal node representing an identifier. These nodes have a type that extends the type corresponding to the nonterminal on the left hand side.

Object oriented languages give an elegant, if somewhat verbose way of specifying the attributes associated with the nodes of the abstract syntax tree.

Sample Input and Output

The sample input

```
(a-b)-c
a-(b-c)
a*b+c*d
(a*b)+(c*d)
(a+b)*(c+d)
f(a,b,c,d)
f(a+b,c*d)
```

generates the output

```
a - b - c
a - ( b - c )
a * b + c * d
a * b + c * d
( a + b ) * ( c + d )
f( a, b, c, d )
f( a + b, c * d )
```

The effects of omitting the precedence declarations

What happens if we omit the precedence relations?

Consider the following state, corresponding to having seen "... Expr – Expr".

```
lalr_state [26]: {
  [Expr ::= Expr (*) INCR ,
     {RIGHT NEWLINE PLUS MINUS STAR DIVIDE ASSIGN INCR DECR COMMA }]
  [Expr ::= Expr (*) DIVIDE Expr ,
     {RIGHT NEWLINE PLUS MINUS STAR DIVIDE ASSIGN INCR DECR COMMA }]
  [Expr ::= Expr (*) PLUS Expr ,
     {RIGHT NEWLINE PLUS MINUS STAR DIVIDE ASSIGN INCR DECR COMMA }]
  [Expr ::= Expr (*) STAR Expr ,
     {RIGHT NEWLINE PLUS MINUS STAR DIVIDE ASSIGN INCR DECR COMMA }]
  [Expr ::= Expr (*) DECR ,
     {RIGHT NEWLINE PLUS MINUS STAR DIVIDE ASSIGN INCR DECR COMMA }]
  [Expr ::= Expr (*) ASSIGN Expr ,
     {RIGHT NEWLINE PLUS MINUS STAR DIVIDE ASSIGN INCR DECR COMMA }]
  [Expr ::= Expr MINUS Expr (*) ,
     {RIGHT NEWLINE PLUS MINUS STAR DIVIDE ASSIGN INCR DECR COMMA }]
  [Expr ::= Expr (*) MINUS Expr ,
     {RIGHT NEWLINE PLUS MINUS STAR DIVIDE ASSIGN INCR DECR COMMA }]
}
transition on ASSIGN to state [23]
transition on DIVIDE to state [22]
```

```
transition on MINUS to state [21]
transition on INCR to state [20]
transition on PLUS to state [19]
transition on STAR to state [18]
transition on DECR to state [17]
```

We have "... Expr - Expr" on the top of stack. We want to reduce if we get a "+", "-" or lower precedence operator, but shift if we get a "*" or "/", or higher precedence operator.

```
From state #26
     RIGHT:REDUCE(8) NEWLINE:REDUCE(8) PLUS:REDUCE(8)
     MINUS:REDUCE(8) STAR:SHIFT(18) DIVIDE:SHIFT(22)
     ASSIGN:REDUCE(8) INCR:SHIFT(20) DECR:SHIFT(17)
     COMMA:REDUCE(8)
```

But, omitting the precedences gives us shifts for all operators.

```
From state #26
     RIGHT:REDUCE(8) NEWLINE:REDUCE(8) PLUS:SHIFT(19)
     MINUS:SHIFT(21) STAR:SHIFT(18) DIVIDE:SHIFT(22)
     ASSIGN:SHIFT(23) INCR:SHIFT(20) DECR:SHIFT(17)
     COMMA:REDUCE(8)
```

In other words, the parser misbehaves.

So you can't just ignore conflicts.

CUP generates error messages:

```
*** Shift/Reduce conflict found in state #26
  between Expr ::= Expr MINUS Expr (*)
  and     Expr ::= Expr (*) PLUS Expr
  under symbol PLUS
  Resolved in favor of shifting.

*** Shift/Reduce conflict found in state #26
  between Expr ::= Expr MINUS Expr (*)
  and     Expr ::= Expr (*) MINUS Expr
  under symbol MINUS
  Resolved in favor of shifting.

*** Shift/Reduce conflict found in state #26
  between Expr ::= Expr MINUS Expr (*)
  and     Expr ::= Expr (*) STAR Expr
  under symbol STAR
  Resolved in favor of shifting.

*** Shift/Reduce conflict found in state #26
  between Expr ::= Expr MINUS Expr (*)
  and     Expr ::= Expr (*) DIVIDE Expr
  under symbol DIVIDE
  Resolved in favor of shifting.

*** Shift/Reduce conflict found in state #26
  between Expr ::= Expr MINUS Expr (*)
  and     Expr ::= Expr (*) ASSIGN Expr
  under symbol ASSIGN
  Resolved in favor of shifting.

*** Shift/Reduce conflict found in state #26
  between Expr ::= Expr MINUS Expr (*)
  and     Expr ::= Expr (*) INCR
  under symbol INCR
  Resolved in favor of shifting.
```

```
*** Shift/Reduce conflict found in state #26
  between Expr ::= Expr MINUS Expr (*)
  and     Expr ::= Expr (*) DECR
  under symbol DECR
  Resolved in favor of shifting.
```