The Language from the Dragon Book in ANTLR

Stephen A. Edwards
Columbia University

1 Introduction

The second edition of the Dragon book[1] describes the implementation of a compiler for a little imperative language. This compiler, described in Appendix A, is written entirely in Java with a handwritten lexical analyzer and recursive-descent parser and produces three-address code.

Here, I implement the same language using the ANTLR compiler generator tool[2]. The Java code for the scanner and parser are now generated by ANTLR and the structure of the compiler has changed slightly. Whereas in the original implementation the parser directly generated the IR (e.g., Stmt objects), this implementation first generates a generic AST using ANTLR's built-in facility for this, then uses a "tree walker" to translate this AST into the IR. While the ANTLR-generated parser could be written to also generate the IR, I consider this a preferred style since it greatly clarifies the specification of the grammar.

The syntax of the language is a stripped-down version of C. Figure 1 shows a small (nonsensical) program and the corresponding output from the compiler.

The grammar of the language is shown in Figure 2. This differs slightly from the grammar given in Appendix A: it also allows the empty statement (i.e., a lone semicolon). The Parser.java file in Appendix A also supports such a statement.

2 The Scanner

The scanner specification is straightforward. It sets the lookahead to two characters ("options { k = 2; }") to distinguish operators such as ">" from ">=."

The rule for whitespace updates the current line number (implicitly assuming Unix-style line termination, i.e., the newline character alone) and discards the token ("$setType(Token.SKIP);").

The *NUM* rule uses a trick. Since floating-point literals are distinguished from integers by the presence of a decimal point, the rule sets the token type to *REAL* if it encounters a decimal point.

grammar.g (part 1)

```
class MyLexer extends Lexer;
options { k = 2; }

WHITESPACE : (' ' | '\t' | '\n' { newline(); } )+  { $setType(Token.SKIP); } ;

protected DIGITS : ('0'..'9')+ ;

NUM : DIGITS ('.' DIGITS { $setType(REAL); } )? ;

AND :    "&&" ;   LE :     "<=" ;   SEMI :   ';'  ;
OR  :    "||" ;   GT :     '>'  ;   LPAREN : '('  ;
ASSIGN : '='  ;   GE :     ">=" ;   RPAREN : ')'  ;
EQ :     "==" ;   LBRACE : '{'  ;   PLUS :   '+'  ;
NOT :    '!'  ;   RBRACE : '}'  ;   MINUS :  '-'  ;
NE :     "!=" ;   LBRACK : '['  ;   MUL :    '*'  ;
LT :     '<'  ;   RBRACK : ']'  ;   DIV :    '/'  ;

ID : ('_' | 'a'..'z' | 'A'..'Z') ('_' | 'a'..'z' | 'A'..'Z' | '0'..'9')* ;
```

[1] Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 2006. Second edition.

[2] www.antlr.org

```
{
  int i; int j; float[10][10] a;
  i = 0;
  while ( i < 10 ) {
    j = 0;
    while ( j < 10 ) {
      a[i][j] = 0;
      j = j+1;
    }
    i = i+1;
  }
  i = 0;
  while ( i < 10 ) {
    a[i][i] = 1;
    i = i+1;
  }
}
```

(a)

```
L1:       i = 0
L3:       iffalse i < 10 goto L4
L5:       j = 0
L6:       iffalse j < 10 goto L7
L8:       t1 = i * 80
          t2 = j * 8
          t3 = t1 + t2
          a [ t3 ] = 0
L9:       j = j + 1
          goto L6
L7:       i = i + 1
          goto L3
L4:       i = 0
L10:       iffalse i < 10 goto L2
L11:       t4 = i * 80
          t5 = i * 8
          t6 = t4 + t5
          a [ t6 ] = 1
L12:       i = i + 1
          goto L10
L2:
```

(b)

Figure 1: (a) A program written in the little language. (b) The output of the compiler.

program	→	*block*
block	→	{ *decls stmts* }
decls	→	*decls decl* | ε
decl	→	*type* **id** ;
type	→	*type* [**num**] | **basic**
stmts	→	*stmts stmt* | ε
stmt	→	*loc* = *bool* ;
	|	**if** (*bool*) *stmt*
	|	**if** (*bool*) *stmt* **else** *stmt*
	|	**while** (*bool*) *stmt*
	|	**do** *stmt* **while** (*bool*) ;
	|	**break** ;
	|	*block*
	|	;
loc	→	*loc* [*bool*] | **id**
bool	→	*bool* || *join* | *join*
join	→	*join* && *equality* | *equality*
equality	→	*equality* == *rel* | *equality* != *rel* | *rel*
rel	→	*expr* < *expr* | *expr* <= *expr* | *expr* >= *expr* | *expr* > *expr* | *expr*
expr	→	*expr* + *term* | *expr* − *term* | *term*
term	→	*term* * *unary* | *term* / *unary* | *unary*
unary	→	! *unary* | − *unary* | *factor*
factor	→	(*bool*) | *loc* | **num** | **real** | `true` | `false`

Figure 2: The grammar for the little language from the Dragon Book.

3 The Parser

The parser follows the structure of the grammar in Figure 2. Automatic AST building is enabled ("options { buildAST=true; }") and two additional token types are declared.

The rule for *program* builds a subtree rooted at a left brace. Its children are DECLS-rooted subtree and the statements in the block. The *decls* rule adds the DECLS node, which has no syntactic analog in the source program.

I rewrote the rules for declarations to avoid left-recursion. They still consist of a basic type name followed by zero or more bracketed array dimensions and an identifier.

The *stmt* rule is almost identical to the rule from the grammar. Certain keywords (e.g., *if*, *while*) are marked to become subtree roots, and the standard *else* ambiguity is resolved by marking the optional *else* term as greedy).

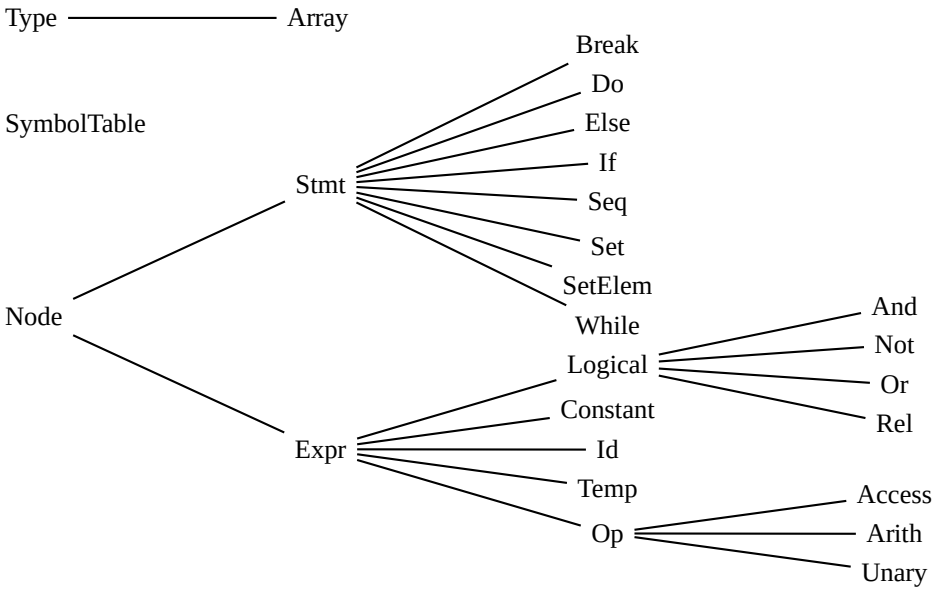
There is one rule per level of expression precedence. Each is coded in the usual way in ANTLR: with right-iteration instead of left-recursion. Although this may appear to make them associate in the wrong way, it actually builds the desired right-leaning tree.

The *unary* rule turns a unary minus sign into a "NEGATE" token to avoid the ambiguity in the AST between it and subtraction.

grammar.g (part 2)

```
class MyParser extends Parser;
options { buildAST = true; }
tokens  { NEGATE; DECLS; }

program : LBRACE^ decls (stmt)* RBRACE! ;

decls : (decl)*  { #decls = #([DECLS, "DECLS"], #decls); } ;

decl : ("int" | "char" | "bool" | "float") (LBRACK! NUM RBRACK!)* ID SEMI! ;

stmt : loc ASSIGN^ bool SEMI!
     | "if"^ LPAREN! bool RPAREN! stmt (options {greedy=true;}: "else"! stmt)?
     | "while"^ LPAREN! bool RPAREN! stmt
     | "do"^ stmt "while"! LPAREN! bool RPAREN! SEMI!
     | "break" SEMI!
     | program
     | SEMI
     ;

loc      : ID^ (LBRACK! bool RBRACK!)* ;
bool     : join (OR^ join)* ;
join     : equality (AND^ equality)* ;
equality : rel ((EQ^ | NE^) rel)* ;
rel      : expr ((LT^ | LE^ | GT^ | GE^) expr)* ;
expr     : term ((PLUS^ | MINUS^) term)* ;
term     : unary ((MUL^ | DIV^) unary)* ;
unary    : MINUS^ unary { #unary.setType(NEGATE); } | NOT^ unary | factor ;
factor   : LPAREN! bool RPAREN! | loc | NUM | REAL | "true" | "false" ;
```

4 The IR Classes



The IR classes are adapted from those in the Dragon Book. The main change is that the classes in the Dragon book use instances of the *Word* class (an extension of the *Token* class their scanner produces), whereas mine just use *String*s.

4.1 The SymbolTable Class

The SymbolTable class stores named identifiers, allows them to be located by name, and keeps track of its parent. Its functionality is equivalent to the *Env* class in the Dragon Book except that I index on *String*s; they use their own *Token* class.

SymbolTable.java

```
import java.util.*;

public class SymbolTable {
    private Hashtable table;
    protected SymbolTable outer;
    public SymbolTable(SymbolTable st) {
        table = new Hashtable();
        outer = st;
    }
    public void put(String token, Type t, int b) {
        table.put(token, new Id(token, t, b));
    }
    public Id get(String token) {
        for (SymbolTable tab = this ; tab != null ; tab = tab.outer) {
            Id id = (Id)(tab.table.get(token));
            if ( id != null ) return id;
        }
        return null;
    }
}
```

4.2 The Type Classes

The *Type* and *Array* classes represent types of expressions and local variables. Each type has a name and an integer width (size) in bytes. The class defines four unique type objects for the four fundamental types. The *numeric* method reports whether a type is a char, int, or float (i.e., not a bool or array). The *max* method returns the "largest" type required to combine the results of an arithmetic operation on two numeric types. Specifically, chars are promoted to ints, and ints are promoted to floats.

Type.java

```
public class Type {
    public int width = 0;
    public String name = "";
    public Type(String s, int w) { name = s; width = w; }
    public static final Type
        Int   = new Type("int", 4),
        Float = new Type("float", 8),
        Char  = new Type("char", 1),
        Bool  = new Type("bool", 1);
    public static boolean numeric(Type p) {
        return p == Type.Char || p == Type.Int || p == Type.Float;
    }
    public static Type max(Type p1, Type p2) {
        if (!numeric(p1) || !numeric(p2)) return null;
        else if (p1 == Type.Float || p2 == Type.Float) return Type.Float;
        else if (p1 == Type.Int || p2 == Type.Int) return Type.Int;
        else return Type.Char;
    }
}
```

The *Array* class represents array types. In addition to a name and width, it holds which type it is an array of and its size.

Array.java

```
public class Array extends Type {
    public Type of;
    public int size = 1;
    public Array(int sz, Type p) {
        super("[]", sz * p.width);
        size = sz;
        of = p;
    }
    public String toString() { return "[" + size + "] " + of.toString(); }
}
```

4.3 The Node Classes

All intermediate representation classes extend the *Node* class. It contains a simple mechanism for generating unique labels and two simple printing methods.

Node.java

```
public class Node {
    void error(String s) { throw new Error(s); }
    static int labels = 0;
    public static int newlabel() { return ++labels; }
    public static void emitlabel(int i) { System.out.print("L" + i + ":"); }
    public static void emit(String s) { System.out.println("\t" + s); }
}
```

4.4 The Expr Classes

Expressions are represented by objects that extend the *Expr* class. It contains a string, which is used to represent identifiers, constants, and operators, and a type, which represents the type of the expression itself.

The *reduce* method, overridden in the *Op* subclass, generates code that computes an expression down to a single address: a *Constant*, an *Id*, or a *Temp*.

The *gen* method, usually overridden to change the arguments of an expression to reduced version, returns a "term" that can appear on the right side of a three-address operation. For example, if an expression is $E_1 + E_2$, it returns an expression of the form $e_1 + e_2$, where e_1 and e_2 are addresses.

The *jumping* method, overridden in the lazy logical operators to transform Boolean operations into tests and jumps. It takes two label indices t and f and generates code that

branches to *t* if the (assumed to be Boolean) expression is true and to *f* otherwise. The zero label is treated specially: as a fall-through to the next instruction in sequence.

The *emitjumps* method generates code for an if-then-else construct given an argument to test and two label indices.

The *toString* method returns a textual representation of the expression; it is overridden in some derived classes.

Expr.java

```
public class Expr extends Node {
    public String s;
    public Type type;
    Expr(String tok, Type p) { s = tok; type = p; }
    public Expr gen() { return this; }
    public Expr reduce() { return this; }
    public void jumping(int t, int f) { emitjumps(toString(), t, f); }
    public void emitjumps(String test, int t, int f) {
        if (t != 0 && f != 0) {
            emit("if " + test + " goto L" + t);
            emit("goto L" + f);
        } else if (t != 0) emit("if " + test + " goto L" + t);
        else if (f != 0) emit("iffalse " + test + " goto L" + f);
    }
    public String toString() { return s; }
}
```

The *Id* class adds an offset field, which holds the address of the identifier relative to the frame pointer.

Id.java

```
public class Id extends Expr {
    public int offset;
    public Id(String id, Type p, int b) { super(id,p); offset = b; }
}
```

The *Constant* class represents integer, floating-point, and Boolean literals. It defines global instances of the two Boolean literals, and overrides the *jumping* method that checks whether the value of the constant is explicitly true or false.

Constant.java

```
public class Constant extends Expr {
    public Constant(String tok, Type p) { super(tok,p); }
    public Constant(int i) { super("" + i, Type.Int); }
    public static final Constant
        True = new Constant("true", Type.Bool),
        False = new Constant("false", Type.Bool);
    public void jumping(int t, int f) {
        if (this == True && t != 0) emit("goto L" + t);
        else if (this == False && f != 0) emit("goto L" + f);
    }
}
```

The *Temp* class represents temporary variables needed for evaluating complex expressions. It introduces a global variable *count* that ensures the name of each temporary is unique and overrides the *toString* method.

Temp.java

```
public class Temp extends Expr {
    static int count = 0;
    int number;
    public Temp(Type p) { super("t", p); number = ++count; }
    public String toString() { return "t" + number; }
}
```

4.4.1 The Op Classes

Arithmetic operators extend the *Op* class. The *reduce* method creates a new temporary and copies the result of the operator into it.

Op.java

```
public class Op extends Expr {
    public Op(String tok, Type p) { super(tok,p); }
    public Expr reduce() {
        Expr x = gen();
        Temp t = new Temp(type);
        emit(t.toString() + " = " + x.toString());
        return t;
    }
}
```

The *Access* class represents an indexed access into an array. Its *gen* method *reduce*s the index to a simple address.

Access.java

```
public class Access extends Op {
    public Id array;
    public Expr index;
    public Access(Id a, Expr i, Type p) { super("[]", p); array = a; index = i; }
    public Expr gen() { return new Access(array, index.reduce(), type); }
    public void jumping(int t, int f) { emitjumps(reduce().toString(), t, f); }
    public String toString() {
        return array.toString() + " [ " + index.toString() + " ]";
    }
}
```

The *Arith* class represents binary arithmetic operators such as $+$. It holds the operator as a string in the *Expr* class's *s* field, its constructor verifies that the two expressions are type-consistent, and its *gen* method calls *reduce* on both child expressions.

Arith.java

```
public class Arith extends Op {
    public Expr expr1, expr2;
    public Arith(String op, Expr x1, Expr x2) {
        super(op, null); expr1 = x1;  expr2 = x2;
        type = Type.max(expr1.type, expr2.type);
        if (type == null) error("type error");
    }
    public Expr gen() { return new Arith(s, expr1.reduce(), expr2.reduce()); }
    public String toString() {
        return expr1.toString() + " " + s + " " + expr2.toString();
    }
}
```

The *Unary* class represents unary negation. Like *Arith*, it checks the type of its operand and calls *reduce* in its *gen* method.

Unary.java

```
public class Unary extends Op {
    public Expr expr;
    public Unary(String tok, Expr e) {
        super(tok, null); expr = e;
        type = Type.max(Type.Int, expr.type);
        if (type == null) error("type error");
    }
    public Expr gen() { return new Unary(s, expr.reduce()); }
    public String toString() { return s + " " + expr.toString(); }
}
```

4.4.2 The Logical Classes

All the Boolean operators are derived from the *Logical* class. The constructor checks the types of the arguments using the *check* method (overridden in derived classes). The *gen* method uses the *jumping* method, which evaluates a Boolean expression and sends control to true and false targets, to put either true or false into a temporary.

Logical.java

```
public class Logical extends Expr {
    public Expr expr1, expr2;
    Logical(String tok, Expr x1, Expr x2) {
        super(tok, null); expr1 = x1; expr2 = x2;
        type = check(expr1.type, expr2.type);
        if (type == null) error("type error");
    }
    public Type check(Type p1, Type p2) {
        if ( p1 == Type.Bool && p2 == Type.Bool ) return Type.Bool;
        else return null;
    }
    public Expr gen() {
        int f = newlabel();
        int a = newlabel();
        Temp temp = new Temp(type);
        this.jumping(0, f);
        emit(temp.toString() + " = true");
        emit("goto L" + a);
        emitlabel(f);
        emit(temp.toString() + " = false");
        emitlabel(a);
        return temp;
    }
    public String toString(){
        return expr1.toString() + " " + s + " " + expr2.toString();
    }
}
```

The *Not* class handles the unary not operator (!). Notable is its *jumping* method, which simply swaps false and true target destinations.

Not.java

```
public class Not extends Logical{
    public Not(Expr x2) { super("!", x2, x2); }
    public void jumping(int t, int f) { expr2.jumping(f, t); }
    public String toString() { return s + " " + expr2.toString(); }
}
```

The *And* and *Or* classes is for the logical and and or operators. Their *jumping* methods effectively generates nested *if-else* constructs.

And.java

```
public class And extends Logical{
    public And(Expr x1, Expr x2) { super("&&", x1, x2); }
    public void jumping(int t, int f) {
        int label = f != 0 ? f : newlabel();
        expr1.jumping(0, label);
        expr2.jumping(t, f);
        if (f == 0) emitlabel(label);
    }
}
```

Or.java

```
public class Or extends Logical {
    public Or(Expr x1, Expr x2) { super("||", x1, x2); }
    public void jumping(int t, int f){
        int label = t != 0 ? t : newlabel();
        expr1.jumping(label, 0);
        expr2.jumping(t, f);
        if ( t == 0 ) emitlabel(label);
    }
}
```

The *Rel* class handles relational operators (==, >=, etc.). Its *jumping* method evaluates both child expressions, performs the comparison, puts the results into a temporary, and finally calls emitjumps to send control to the true and false targets.

Rel.java

```
public class Rel extends Logical {
    public Rel(String tok, Expr x1, Expr x2) { super(tok, x1, x2); }
    public Type check(Type p1, Type p2) {
        if (p1 instanceof Array || p2 instanceof Array) return null;
        else if (p1 == p2) return Type.Bool;
        else return null;
    }
    public void jumping(int t, int f){
        Expr a = expr1.reduce();
        Expr b = expr2.reduce();
        String test = a.toString() + " " + s + " " + b.toString();
        emitjumps(test, t, f);
    }
}
```

4.5 The Stmt Classes

All the statement-specific classes are derived from the *Stmt* class. Like *Expr*, it has a *gen* method, but the one for *Stmt* does not return anything (a statement only has side-effects; it does not does not produce a result, just side-effects) and instead takes two arguments: *a*: a label for the code immediately after the statement and *b*: a label for the start of the statement itself.

The *Null* static member is intended to be a unique empty statement. It is used, for example, to represent the empty statement. (;)

The *Enclosing* global variable is used by the *Break* class as the target for the *break* instruction. E.g., within a *while* statement, the *Enclosing* variable points to the statement for the *while* and sends control to immediately after it.

The *after* field is used to record the index of the label immediately after the statement.

Stmt.java

```
public class Stmt extends Node {
    public Stmt() {}
    public static Stmt Null = new Stmt();
    public void gen(int b, int a) {}
    int after = 0;
    public static Stmt Enclosing = Stmt.Null;
}
```

The *Break* class holds the statement it escapes from in the *stmt* field, captures this from the *Enclosing* field of the *Stmt* class in its constructor, and generates a *goto* statement to just after this command in its *gen* method.

Break.java

```
public class Break extends Stmt {
    Stmt stmt;
    public Break() {
        if (stmt.Enclosing == null) error("unenclosed break");
        stmt = Stmt.Enclosing;
    }
    public void gen(int b, int a) {
        emit("goto L" + stmt.after);
    }
}
```

The *Do* class represents the do-while construct, which consists of a statement body and an expression predicate. These fields are set by the *init* method, which verifies the expression is of Boolean type.

The generated code for a do-while conisists of the body statement followed by an evaluation of the expression that either sends control back to the beginning of the statement (the *b* argument) or falls through.

Do.java

```
public class Do extends Stmt{
    Expr expr;
    Stmt stmt;
    public Do() { expr = null; stmt = null; }
    public void init(Stmt s, Expr x) {
        expr = x;
        stmt = s;
        if (expr.type != Type.Bool) expr.error("boolean required in do");
    }
    public void gen(int b, int a) {
        after = a;
        int label = newlabel();
        stmt.gen(b, label);
        emitlabel(label);
        expr.jumping(b, 0);
    }
}
```

The *While* class handles *white* statements and is much like the *Do* class except that the generated code consists of the test of the expression followed by the body and a jump back to the beginning.

While.java

```
public class While extends Stmt{
    Expr expr;
    Stmt stmt;
    public While() { expr = null; stmt = null; }
    public void init(Expr x,Stmt s){
        expr = x;
        stmt = s;
        if (expr.type != Type.Bool) expr.error("boolean required in while");
    }
    public void gen(int b, int a){
        after = a;
        expr.jumping(0, a);
        int label = newlabel();
        emitlabel(label);
        stmt.gen(label, b);
        emit("goto L" + b);
    }
}
```

The *Else* class handles if-then-else constructs, which contain a predicate expression, a "then" statement, and an "else" statement. The constructor verifies the predicate is of Boolean type and the generated code consists of evaluating the test expression followed by the *then* branch, a jump, and the *else* branch.

Else.java

```
public class Else extends Stmt{
    Expr expr;
    Stmt stmt1,stmt2;
    public Else(Expr x, Stmt s1, Stmt s2){
        expr = x;
        stmt1 = s1;
        stmt2 = s2;
        if (expr.type != Type.Bool) expr.error("boolean required in if");
    }
    public void gen(int b, int a){
        int label1 = newlabel();
        int label2 = newlabel();
        expr.jumping(0, label2);
        emitlabel(label1); stmt1.gen(label1, a); emit("goto L" + a);
        emitlabel(label2); stmt2.gen(label2, a);
    }
}
```

The *If* class is a simplified version of the *Else* class that only has a single statement. The generated code does not need an additional *goto*.

If.java

```
public class If extends Stmt{
    Expr expr;
    Stmt stmt;
    public If(Expr x, Stmt s){
        expr = x;
        stmt = s;
        if (expr.type != Type.Bool) expr.error("boolean required in if");
    }
    public void gen(int b, int a){
        int label = newlabel();
        expr.jumping(0, a);
        emitlabel(label); stmt.gen(label, a);
    }
}
```

The *Seq* class handles statements in sequence. The generated code is the obvious one: the two instructions in sequence separated by a label (so the first statement has a target if it terminates). The two special cases are when one of the two statements is empty.

Seq.java

```
public class Seq extends Stmt {
    Stmt stmt1;
    Stmt stmt2;
    public Seq(Stmt s1, Stmt s2) { stmt1 = s1; stmt2 = s2; }
    public void gen(int b, int a) {
        if (stmt1 == Stmt.Null) stmt2.gen(b, a);
        else if (stmt2 == Stmt.Null) stmt1.gen(b, a);
        else {
            int label = newlabel();
            stmt1.gen(b, label);
            emitlabel(label);
            stmt2.gen(label, a);
        }
    }
}
```

The *Set* class handles assignment statements. Its constructor verifies the type of the expression is consistent with the lvalue (the target of the assignment). It generates code that ends in an assignment, although note that the call to *Expr.gen* may generate a code sequence if the expression is complicated or has side-effects.

Set.java

```
public class Set extends Stmt {
    public Id lvalue;
    public Expr expr;
    public Set(Id i, Expr x) {
        lvalue = i;
        expr = x;
        if (check(lvalue.type, expr.type) == null) error("type error");
    }
    public Type check(Type p1, Type p2) {
        if (p1 instanceof Array || p2 instanceof Array) return null;
        if (p1 == p2) return p2;
        if (Type.numeric(p1) && Type.numeric(p2)) return p2;
        else if (p1 == Type.Bool && p2 == Type.Bool) return p2;
        else return null;
    }
    public void gen(int b, int a) {
        emit(lvalue.toString() + " = " + expr.gen().toString());
    }
}
```

The *SetElem* class handles assignment to array elements. It is like the *Set* class but adds an additional index expression that is reduced before before being made part of the assignment generated by the *gen* method.

SetElem.java

```
public class SetElem extends Stmt {
    public Id array;
    public Expr index;
    public Expr expr;
    public SetElem(Access x, Expr y) {
        array = x.array;
        index = x.index;
        expr = y;
        if (check(x.type, expr.type) == null) error("type error");
    }
    public Type check(Type p1,Type p2) {
        if (p1 instanceof Array || p2 instanceof Array) return null;
        else if (p1 == p2) return p2;
        else if (Type.numeric(p1) && Type.numeric(p2)) return p2;
        else return null;
    }
    public void gen(int b, int a) {
        String s1 = index.reduce().toString();
        String s2 = expr.reduce().toString();
        emit(array.toString() + " [ " + s1 + " ] = " + s2);
    }
}
```

5 The Tree Walker

Now that we have defined all the IR classes, it is finally possible to describe the tree walker, which constructs the IR as a side-effect of its walk.

The tree walker is longer than the parser because it contains a lot of Java code, but its structure is simpler (seven rules vs. thirteen) because it does not have to worry about operator precedence.

It first defines two global variables: *top*, which holds the symbol table for the block currently being analyzed, and *used*, which keeps a count of the the number of bytes allocated in the current function's activation record (i.e., the storage for local variables).

The *program* rule looks for a subtree rooted at an LBRACE token, pushes the topmost symbol table onto the stack (the *program* rule is actually a Java method and *saved_environment* is one of its local variables), creates a new, empty symbol table that points to the old one, deals with the declarations and statements in the block, and finally restores the outermost symbol table.

grammar.g (part 3)

```
class MyWalker extends TreeParser;
{
  SymbolTable top = null;
  int used = 0; // Number of bytes allocated for local declarations
}

program returns [Stmt s]
{ s = null; Stmt s1;}
  : #(LBRACE
      { SymbolTable saved_environment = top; top = new SymbolTable(top); }
      decls
      s=stmts
      { top = saved_environment; }
    )
  ;
```

The rules for *decls*, *type*, and *dims* handle local variable declarations. The *decls* rule looks for a sequence of type/ID pairs under a DECLS token and adds the name/type binding to the active symbol table. The *used* variable is updated with the size of the local variable to keep track of where it will be put in the activation record.

The *type* rule looks for one of the four basic types (bool, char, etc.) followed by an optional sequence of array dimensions. The *dims* rule creates a new *Array* type based on the sequence of sizes it finds. The *dims* rule is written in a right-recursive form so the leftmost dimension appears at the top of the hierarchy.

grammar.g (part 4)

```
decls
{ Type t = null; }
  : #(DECLS
       (t=type ID { top.put(#ID.getText(), t, used); used += t.width; } )* )
  ;

type returns [Type t]
{ t = null; }
  : ( "bool"  { t = Type.Bool;  }
    | "char"  { t = Type.Char;  }
    | "int"   { t = Type.Int;   }
    | "float" { t = Type.Float; } )
    (t=dims[t])?
  ;

dims[Type t1] returns [Type t]
{ t = t1; }
  : NUM (t=dims[t])? { t = new Array(Integer.parseInt(#NUM.getText()), t); }
  ;
```

The *stmts* and *stmt* rules handle sequences and single statements respectively. The *stmts* rule is written in a right-recursive form so it builds a left-leaning tree.

The *stmt* rule handles individual statements. The rules for *while* and *do* push and pop the *Enclosing* statement, which the *break* statement uses as its target. The *program* rule is called recursively to handle block statements (i.e., brace-enclosed statement sequences).

grammar.g (part 5)

```
stmts returns [Stmt s]
{ s = null; Stmt s1; }
  : s=stmt (s1=stmts { s = new Seq(s, s1); } )?
  ;

stmt returns [Stmt s]
{ Expr e1, e2;
  s = null;
  Stmt s1, s2;
}
  : #(ASSIGN e1=expr e2=expr
      { if (e1 instanceof Id) s = new Set((Id) e1, e2);
        else s = new SetElem((Access) e1, e2);
      }
    )
  | #("if" e1=expr s1=stmt
       ( s2=stmt { s = new Else(e1, s1, s2); }
       | /* nothing */ { s = new If(e1, s1); } ))
  | #("while"
       { While whilenode = new While();
         s2 = Stmt.Enclosing;
         Stmt.Enclosing = whilenode; }
       e1=expr
       s1=stmt
       { whilenode.init(e1, s1);
         Stmt.Enclosing = s2;
         s = whilenode; } )
  | #("do"
       { Do donode = new Do();
         s2 = Stmt.Enclosing;
         Stmt.Enclosing = donode; }
       s1=stmt
       e1=expr
       { donode.init(s1, e1);
         Stmt.Enclosing = s2;
         s = donode; } )
  | "break" { s = new Break(); }
  | s=program
  | SEMI { s = Stmt.Null; }
  ;
```

The *expr* rule builds expressions. Note that the structure of the tree already embodies precedence and associativity rules so that a single *expr* rule can handle them all uniformly.

Only the sub-rule for identifiers is complicated. It first checks the symbol table for a matching declaration, giving an error if one is not found. After that, it looks for an optional sequence of expressions corresponding to array indices. If there are any, it builds an expression that calculates the address of the requested entry in the array, using the size information in the *Array* objects it finds on the way. The first dimension is treated specially because it does not have to add its index onto an existing one.

grammar.g (part 6)

```
expr returns [Expr e]
{
  Expr a, b;
  e = null;
}
  : #(OR     a=expr b=expr { e = new Or(a, b); } )
  | #(AND    a=expr b=expr { e = new And(a, b); } )
  | #(EQ     a=expr b=expr { e = new Rel("==", a, b); } )
  | #(NE     a=expr b=expr { e = new Rel("!=", a, b); } )
  | #(LT     a=expr b=expr { e = new Rel("<", a, b); } )
  | #(LE     a=expr b=expr { e = new Rel("<=", a, b); } )
  | #(GT     a=expr b=expr { e = new Rel(">", a, b); } )
  | #(GE     a=expr b=expr { e = new Rel(">=", a, b); } )
  | #(PLUS   a=expr b=expr { e = new Arith("+", a, b); } )
  | #(MINUS  a=expr b=expr { e = new Arith("-", a, b); } )
  | #(MUL    a=expr b=expr { e = new Arith("*", a, b); } )
  | #(DIV    a=expr b=expr { e = new Arith("/", a, b); } )
  | #(NOT    a=expr        { e = new Not(a); } )
  | #(NEGATE a=expr        { e = new Unary("-", a); } )
  | NUM                    { e = new Constant(#NUM.getText(), Type.Int); }
  | REAL                   { e = new Constant(#REAL.getText(), Type.Float); }
  | "true"                 { e = Constant.True; }
  | "false"                { e = Constant.False; }
  | #(ID
       { Id i = top.get(#ID.getText());
         if (i == null) System.out.println(#ID.getText() + " undeclared");
         e = i;
       }
       ( a=expr
         { Type type = e.type;
           type = ((Array)type).of;
           Expr w = new Constant(type.width);
           Expr loc = new Arith("*", a, w);
         }
         ( a=expr
           { type = ((Array)type).of;
             w = new Constant(type.width);
             loc = new Arith("+", loc, new Arith("*", a, w));
           }
         )*
         { e = new Access(i, loc, type); }
       )?
     )
  ;
```

6 The Main class

Finally, the *Main* class invokes the scanner, parser, tree walker, and finally calls *gen* on the topmost statement (i.e., representing the program) to print three-address code, such as that shown in Figure 1(b).

Uncommenting the three lines enables a graphical display of the AST, allowing its structure to be debugged more easily.

Main.java

```
import java.io.*;
// import antlr.debug.misc.*;

class Main {
    public static void main(String[] args) {
        try {
            FileInputStream filename = new FileInputStream(args[0]);
            DataInputStream input = new DataInputStream(filename);
            MyLexer lexer = new MyLexer(new DataInputStream(input));
            MyParser parser = new MyParser(lexer);
            parser.program();
            antlr.CommonAST ast = (antlr.CommonAST) parser.getAST();
            // ASTFrame frame = new ASTFrame("AST", ast);
            // frame.setVisible(true);
            MyWalker walker = new MyWalker();
            Stmt s = walker.program(ast);
            int begin = s.newlabel();
            int after = s.newlabel();
            s.emitlabel(begin);
            s.gen(begin, after);
            s.emitlabel(after);
            System.out.println("");
        } catch (Exception e) {
            System.err.println("exception: " + e);
        }
    }
}
```