17

Practical Parser Writing and Usage

Practical parsing is concerned almost exclusively with context-free (Type 2) and regular (Type 3) grammars. Unrestricted (Type 0) and context-sensitive (Type 1) grammars are hardly used since, first, they are user-unfriendly in that it is next to impossible to construct a clear and readable Type 0 or Type 1 grammar and, second, all known parsers for them have exponential time requirements. Chapter 15 describes a number of polynomial-time and even linear-time parsers for non-Chomsky systems, but few have seen practical application. For more experimental results see (Web)Section 18.2.6.

Regular grammars are used mainly to describe patterns that have to be found in surrounding text. For this application a recognizer suffices. There is only one such recognizer: the finite-state automaton described in Section 5.3. Actual parsing with a regular grammar, when required, is generally done using techniques for CF grammars.

In view of the above we shall restrict ourselves to CF grammars in the rest of this chapter. We start with a comparative survey of the available parsing techniques (Section 17.1). Parsers can be interpretive, table-driven or compiled; the techniques are covered in Section 17.2.1. Section 17.3 presents a simple general context-free parser in Java, both for experimentation purposes and to show the nitty-gritty details of a complete parser. Parsers, both interpretive and compiled, must be written in a programming language; the influence of the programming language paradigm is discussed in Section 17.4. Finally, Section 17.5 exhibits a few unusual applications of the pattern recognition inherent in parsing.

17.1 A Comparative Survey

17.1.1 Considerations

The initial demands on a CF parsing technique are obvious: it should be general (i.e., able to handle all CF grammars), it should be fast (i.e., have linear time requirements) and preferably it should be easy to program. Practically the only way to obtain linear

time requirement is to use a deterministic method; there are non-deterministic methods which work in linear time (for example Bertsch and Nederhof [96]) but there is little practical experience with them.

There are two serious obstacles to this naive approach to choosing a parser. The first is that the automatic generation of a deterministic parser is possible only for a subset of the CF grammars. The second is that, although this subset is often described as "very large" (especially for LR(1) and LALR(1)), experience shows that a grammar that is designed to best describe the language without concern for parsing is virtually never in this set. It is true that for most reasonable grammars a slightly different grammar can be found that generates the same language and that does allow linear-time parsing, but here are two problems with this. Finding such a grammar almost always requires human intervention and cannot be automated. And using a modified grammar has the disadvantage that the resulting parse trees will differ to a certain extent from the ones implied by the original grammar. Furthermore, it is important to notice that no deterministic method can handle ambiguous grammars.

An immediate consequence of the above observations is that the stability of the grammar is an important datum. If the grammar is subject to continual revision, it is impossible or at least highly inconvenient to adapt each version by hand to the requirements of a deterministic method, and we have no choice but to use a general method. Likewise, if the grammar is ambiguous, we should use a general method.

If one has the luxury of being in a position to design the grammar oneself, the choice is simple: design the grammar to be LL(1) and use a predictive recursive descent parser. It can be generated automatically, with good error recovery, and allows semantic routines to be included in-line. This can be summarized as: parsing is a problem only if someone else is in charge of the grammar.

17.1.2 General Parsers

There are three general methods that should be considered: Unger's, Earley's and GLR.

17.1.2.1 Unger

An Unger parser (Section 4.1) is easy to program, especially the form given in Section 17.3.2, but its exponential time requirements limit its applicability to occasional use. The relatively small effort of adding a well-formed substring table (Section 17.3.4) can improve its efficiency dramatically, and in this form it can be very useful, especially if the average input string is limited to some tens of tokens. The thus modified Unger parser requires in principle a time proportional to n^{N+1}, where n is the number of tokens in the input and N is the maximum number of non-terminals in any right-hand side in the grammar, but in practice it is often much faster. An additional advantage of the Unger parser is that it can usually be readily understood by all participants in a project, which is something that can be said of almost no other parser.

17.1.2.2 Earley

A simple, robust and efficient version of the Earley parser has been presented by Graham, Harrison and Ruzzo [23]. It requires a time proportional to n^3 for ambiguous grammars (plus the time needed to enumerate the parse trees), at most n^2 for unambiguous grammars and n for grammars for which a deterministic method would work; in this sense the Earley parser is self-adapting. Since it does not require preprocessing on the grammar, it is possible to have one grammar-independent Earley parser and to supply it with the grammar and the input whenever a parsing is needed. If this is convenient, the Earley parser is preferable to GLR.

17.1.2.3 Generalized LR

At the expense of considerably more programming and some loss of convenience in use, a GLR parser (Section 11.1) will provide a parsing in slightly more than linear time for all but the most ambiguous grammars. Since it requires preprocessing on the grammar, it is convenient to generate a separate parser for each grammar (using a parser generator); if the grammar is, however, very unstable, the preprocessing can be done each time the parser is called. The GLR parser is presently the parser of choice for serious parsing in situations where a deterministic method cannot be applied and the grammar is reasonably stable.

As explained in Section 11.1, a GLR parser uses a table to restrict the breadth-first search and the question arises what type of table would be optimal. Lankhorst [166] determined experimentally that LR(0) and SLR(1) are about equally efficient, and that LALR(1) is about 5-10% faster; LR(1) is definitely worse. So, unless a speed-up of a few percent matters, the simple LR(0) is the recommended parse table.

17.1.2.4 Notes

It should be noted that if any of the general parsers performs in linear time, it may still be a factor of ten or so slower than a deterministic method, due to the much heavier administration they need.

None of the general parsers identifies with certainty a part of the parse tree before the whole parse tree is completed. Consequently, if semantic actions are connected to the grammar rules, none of these actions can be performed until the whole parse is finished. The actions certainly cannot influence the parsing process. They can, however, reject certain parse trees afterwards; this is useful to implement context conditions in a context-free parser.

17.1.3 General Substring Parsers

Although general substring parsing does not differ fundamentally from general full parsing (construct a substring grammar as explained in Section12.1 and use one of the above general CF parsers) a specific substring parser will be much more efficient. Rekers and Koorn [212] and Rekers [213, Chapter 4] describe the details of such a parser.

17.1.4 Linear-Time Parsers

Among the grammars that allow linear-time parsing, the operator-precedence grammars (see Section 9.2.2) occupy a special place, in that they can be ambiguous. They escape the general rule that ambiguous grammars cannot be parsed in linear time by virtue of the fact that they do not provide a full parse tree but rather a parse skeleton. If every sentence in the generated language has only one parse skeleton, the grammar can be operator-precedence. Operator-precedence is by far the simplest practical method; if the parsing problem can be brought into a form that allows an operator-precedence grammar (and that is possible for almost all formula-like inputs), a parser can be constructed by hand in a very short time.

17.1.4.1 Requirements

Now we come to the full linear-time methods. As mentioned above, grammars are not normally in a form that allows deterministic parsing and have to be modified by hand to be so. This implies that for the use of a deterministic parser at least the following conditions must be fulfilled:

- the grammar must be relatively stable, so that the modification process will not have to be repeated too often;
- the user must be willing to accept a slightly different parse tree than would correspond to the original grammar.

Speed is not an issue: any not inordinately long input parses in a fraction of a second with a deterministic parser on a modern machine.

It should again be pointed out that the transformation of the grammar cannot, in general, be performed by a program (if it could, we would have a stronger parsing method).

Leo's improvement of the Earley parser [32] may be a viable alternative. It requires linear time on all deterministic and many other grammars, and does not require preprocessing. Since it is interpreted, we expect it to lose perhaps two orders of magnitude in speed over a table-driven LR parser, but that might not be a problem on present-day machines. Experience with this type of parser is lacking, though.

17.1.4.2 Strong-LL(1) versus LALR(1)

For two deterministic methods, "strong-LL(1)"[1] (Section 8.2.2) and LALR(1) (Section 9.7), parser generators are readily available, both as commercial products and in the public domain. Using one of them will in almost all cases be more practical and efficient than writing your own; for one thing, while writing a parser generator may be (is!) interesting, doing a reasonable job on the error recovery is a protracted affair, not to be taken on lightly. So the choice is between (strong-)LL(1) and

[1] What is advertised as an "LL(1) parser generator" is almost always actually a strong-LL(1) parser generator.

LALR(1). Full-LL(1) or LR(1) are occasionally preferable, and some parser generators for these are available if needed. The main differences between (strong-)LL(1) and LALR(1) can be summarized as follows:

- LL(1) usually requires larger modifications to be made to the grammar than LALR(1).
- LL(1) allows semantic actions to be performed even before the start of an alternative; LALR(1) performs semantic actions only at the end of an alternative.
- LL(1) parsers are often easier to understand and modify.
- If an LL(1) parser is implemented as a recursive-descent parser, the semantic actions can use named variables and attributes, much as in a programming language. No such use is possible in a table-driven parser.
- Both methods are roughly equivalent as to speed and memory requirements; a good implementation of either will outperform a mediocre implementation of the other.

The difference between the two methods disappears largely when the parser yields a complete parse tree and the semantic actions are deferred to a later stage. In such a setup LALR(1) is obviously to be preferred.

People evaluate the difference in power between LL(1) and LALR(1) differently; for some the requirements made by LL(1) are totally unacceptable, others consider them a minor inconvenience, largely offset by the advantages of the method.

If one is in a position to design the grammar along with the parser, there is little doubt that LL(1) is to be preferred: not only will parsing and performing semantic actions be easier, text that conforms to an LL(1) grammar is also clearer to the human reader. A good example is the design of Modula-2 by Wirth (see *Programming in Modula-2 (Third, corrected edition)* by Niklaus Wirth, Springer-Verlag, Berlin, 1985).

17.1.4.3 Table Size

The table size of a deterministic parser is moderate, from 10K to 100K bytes for the deterministic parsers to megabytes for the non-canonical ones, and will be a problem in very few applications. The strongest linear-time method with negligible table size is weak precedence with precedence functions.

17.1.5 Linear-Time Substring Parsers

On the subject of practical linear-time substring parsers there is little difficulty; Bates and Lavie's [214] is the prime candidate. It is powerful, efficient and not too difficult to implement.

17.1.6 Obtaining and Using a Parser Generator

The approach is simple: search the Internet. There are a surprising number of parser generators out there, including some for general CF parsing, based both on Earley

and on GLR. We shall not give names or URLs here, since, as we said in the Preface, such information is ephemeral and the best URL is a few well-chosen search terms submitted to a good Web search engine.

For advanced handling of the parse tree there are a number of parse tree processor generator tools and prettyprinters for parse trees. Converting the parse tree to XML and using a Web browser to view it is a simple but viable alternative for the latter.

17.2 Parser Construction

Parsing starts with a grammar and an input string (A) supplied by the user and ends with the result (B) desired by the user. Even disregarding the actual parsing technique used, there are several ways to get from A to B. Also, actually there is more than one destination B: generally the user will want a structuring of the input, in the form of a parse tree or a parse grammar, to be processed further, but sometimes it is the semantics of the input that is desired immediately. An example is a parser for arithmetic expressions like `4+5*6+8` which is expected to produce the answer `42` directly. Another is a PostScript or HTML interpreter, where most of the input file is executed while it is being parsed. See Clark [355] for more about the difference between building a parse tree and having immediate semantics.

17.2.1 Interpretive, Table-Based, and Compiled Parsers

Just as a program in a programming language like C or Java can either be interpreted or compiled into a binary executable, a grammar can either be interpreted or compiled into a parser. But we have to be careful of what we mean by these words. Actually it is the combination of program and input which is interpreted, and it is the program only that is compiled into executable code. Likewise the combination of grammar and input can be interpreted (as shown in Figure 17.1), and the grammar on its own can be compiled into a parser by a parser generator (Figure 17.2). In both pictures the boxes marked with a 'U' in the top left corner represent files supplied by the parser user; those with 'P' are created by the parser writer; those with a $\times$ are executable programs; and unmarked boxes represent generated files.

Figure 17.1 is simple: the source code of the interpreter is fed through a compiler, which produces the actual interpreter. It is then given both the user grammar and ditto input, and does its work. Interpreters read the grammar every time a new input is offered; this has the advantage that always the most up-to-date version of the grammar is used. They are also often easier to write than parser generators. Interpreters have the disadvantage of being slow, both because of the overhead inherent in interpreting, and because the grammar is processed time and again. It is sometimes convenient to incorporate the grammar as static data in the interpreter, thus creating a parser specific to that grammar.

Figure 17.2, concerning compiled parsers, is more complicated. First the code of the parser generator is compiled into the parser generator proper; this happens out of sight of the user. The user then supplies the grammar to the parser generator, which

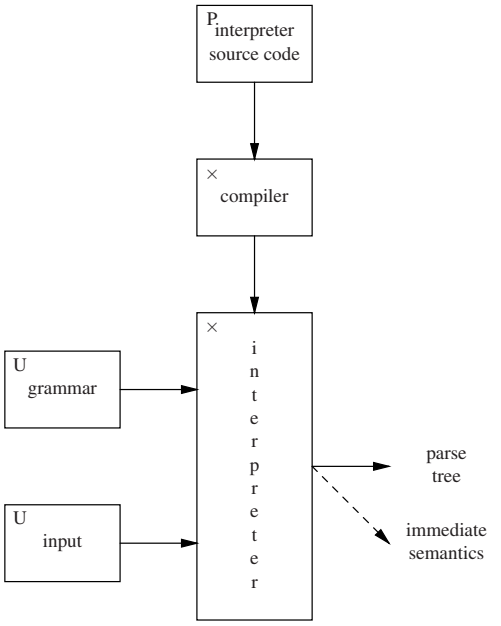

Fig. 17.1. Use of an interpreting parser (U = user-supplied; P = parser writer supplied)

turns it either into parse tables or into parser code, depending on the nature of the parser generator. The result is then compiled into the parser proper; if the parser is table-driven, a driver is added. The parser can then be used with different inputs as often as needed.

There is unfortunately no standard terminology to distinguish the three types; we shall call them "interpreters" (or "interpreting parsers"), "table-driven parsers", and "compiled parsers", respectively.

17.2.2 Parsing Methods and Implementations

If the parser is complicated it is usually easier to write it as an interpreter rather than to generate code for it. So most of the general CF parsers are programmed as interpreters and the situation in Figure 17.1 applies. Since general CF parsing is usually done in fairly experimental circumstances, in which the grammar is equally likely to change as the input, this has the additional advantage that one is not penalized for changing the grammar.

Compiling a general CF parser into code is not impossible, however; Aycock and Horspool [37] present a compiled Earley parser.

LL(1) parsers come in two varieties: compiled, using recursive descent (Section 8.2.6), in which a procedure is created for each non-terminal; and table-driven, using a table like the one in Figure 8.10 and a pushdown automaton as described in Section 6.2. The recursive descent version is probably more usual.

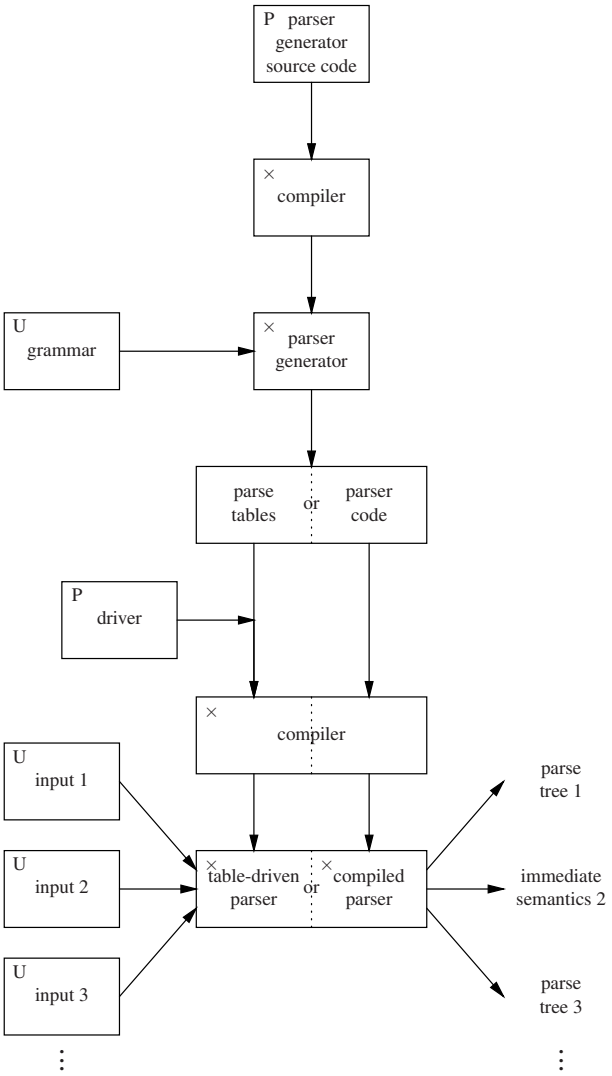

Fig. 17.2. Use of a compiled parser

Almost all LR parsers are table-driven, using tables like the one in Figure 9.28, and a pushdown automaton. Examples of compiled LR parsers are discussed by Pennello [70], Horspool and Whitney [85] and Bhamidipaty and Proebsting [354]. The recursive ascent parsers (Section 9.10.6) also lead to compiled LR parsers.

There is ample evidence that compiled parsers are faster than table-driven parsers, in addition to having the advantage that semantic routines can be included conveniently; see, for example, Waite and Carter [339], and the above papers. At least two causes for this phenomenon have been identified: we avoid the time re-

quired to identify the action to be performed and to prepare for calling it; and the often superb optimization done by present-day compilers.

An interesting possibility is to start the parser as an interpreter but to memoize the table information from every parsing decision made and reuse it when the same decision comes up again. This is known as *lazy table construction*. It is especially effective for lexical analysers, but can also help in modularizing the parsing process (Koskimies [342]). It has the possible disadvantage that only the activated part of the grammar gets checked.

For finite-state automata table-driven implementations are the norm. But see Jones [146] for compiled FS automata.

17.3 A Simple General Context-Free Parser

Although LL(1) and LALR(1) parsers are easy to come by, they are of limited use outside the restricted field of programming language processing. General CF parsers are available, but are often complicated, both to understand and to use. We will therefore present here in full detail a simple general parser that will yield all parsings of a sentence according to a CF grammar, with no restriction imposed on the grammar. It is small, written in Java, and enables the reader to experiment directly with a general CF parser that is under his or her full control. The parser described in the Sections 17.3.1 to 17.3.3 takes exponential time in the worst case; a memoization feature which reduces the time requirement to polynomial is discussed in Section 17.3.4. The interested reader who has access to a Prolog interpreter may wish to look into DCGs ("Definite Clause Grammars", Section 6.7). These may be more useful than the parser in this chapter since they allow context conditions to be applied, but they cannot handle left recursion unless special measures are taken, as for example those in Section 6.8).

17.3.1 Principles of the Parser

The parser, presented as a set of Java classes in Sections 17.3.2 to 17.3.4, is the simplest we can think of that puts no restrictions on the grammar. Since it searches a forest of possible parse trees to find the applicable ones, it is not completely trivial, though. The parser is an Unger parser in that it does a top-down analysis, dividing the input into segments that are to be matched to symbols in the pertinent right-hand sides. A depth-first search, using recursive descent, is used to enumerate all possibilities.

To avoid clutter, not all class declarations are shown in the printings of class declarations, and many declarations of administrative methods (straightforward constructors, `toString()`, etc.) are suppressed; the full working parser can be downloaded from this book's web site.

17.3.2 The Program

The central objects in the parser are **Goal**s, **RuleGoal**s, and **DottedGoal**s; they form a small linear hierarchy. A **Goal** contains a non-terminal **lhs**, a position in the input **pos**, and a length **length**; it represents an attempt to derive the indicated segment of the input from the non-terminal. **RuleGoal** is a subclass of **Goal**, in which the non-terminal has been narrowed down to a specific rule, **rule**. A **DottedGoal** is a **RuleGoal** with dots in the right hand side and the input segment; the positions of the dots are given as two integer fields **rhsUsed**, and **inputUsed**. A **DottedGoal** succeeds when it can match the remainder of the right-hand side to the remainder of the input segment.

The recursion in Java is used to search the space of all alternatives of non-terminals and of all possible segment lengths. The **DottedGoal**s generated by this search are put on a stack called **DottedGoalStack**. The **DottedGoalStack** starts off with a **DottedGoal** containing the start symbol and the entire input; whenever the stack becomes empty, a parsing has been found. To print a listing of the rules that led to the parsing, newly tried rules are stacked on a **RuleStack**, and removed from it when no more matchings for them can be found.[2]

The **DottedGoalStack** contains the active nodes in the parse tree, which is only a fraction of the nodes of the parse tree as already recognized (Figure 6.2). The leftmost derivation of the parse tree as far as recognized can be found on the stack **RuleStack**. When the **DottedGoalStack** becomes empty, a complete parsing has been found, recorded in the **RuleStack**.

The main class of the parser, shown in Figure 17.3 just loads the grammar from

```
public class TopDownParser {
    public static void main(String[] args) {
        String userGrammarName = "UserGrammar4";
        Grammar.load(userGrammarName);
        Grammar.parse();
    }
}
```

Fig. 17.3. The driver

a user class and parses the offered strings. The demo grammar file **UserGrammar4** specifies the grammar

```
S  ->  LSR | ε
L  ->  ( | ε
R  ->  )
```

[2] Somebody who would suggest that we are implementing a Prolog interpreter in disguise would be right.

This grammar forces the parser to match either an opening parenthesis or ε to each closing parenthesis, which makes most inputs very ambiguous. The demo applies it to the strings `())` and `(((())))`.

The class **`Grammar`** (Figure 17.4) uses a call to

```
import java.util.ArrayList;
public class Grammar {
    private static ArrayList<Rule> ruleList = new ArrayList<Rule>();
    public static Symbol startSym = null;
    public static Rule getRule(int n) {return ruleList.get(n);}
    public static void load(String filename) {
        ReadGrammar.readGrammar(filename);
    }
    public static void parse() {
        Input s;
        while ((s = ReadInput.readInput()) != null) TD.parse(s);
    }
}
```

Fig. 17.4. The class **`Grammar`**

`ReadGrammar.readGrammar(String filename)` (not shown) to read the grammar from a file and store its start symbol in **`startSym`** and its rules in the array list **`ruleList`**. The rules are objects of class **`Rule`** (not shown), each of which has two public fields, **`lhs`** and **`rhs`**. The tokens in the grammar and the input are objects of a class **`Symbol`** (not shown); this allows any lexical analyser to be plugged in independently.

The class **`Grammar`** also supplies the method **`parse()`** used above in **`TopDownParser`**. It reads a sequence of input strings and calls the top-down parser in class **`TD`** for each of them.

This brings us to the class **`TD`** (Figure 17.5), which supplies the methods **`parse(Input)`** and **`parsingFound()`**, and counts the number of derivations. The method **`parse(Input)`** starts by doing some initializations, which include creating a new **`RuleStack`** (not shown) and a new **`DottedGoalStack`** (not shown), and clearing the derivation count; for **`knownRuleGoals`** see the next paragraph. Next it prints the **`input`** in a message. Then the real parsing starts: **`parse(Input)`** creates a **`Goal`** consisting of the start symbol of the grammar, the start position 0, and the length of the input, and activates it by calling its method **`doWork()`**. When the activation returns, all parsings have been counted and reported. The method **`parsingFound()`** counts each parsing and reports it by printing the rule stack.

To prepare the way for a system to memoize known parsings, calls to methods in an object of class **`knownRuleGoals`** have already been placed in the presented code. We shall ignore them until Section 17.3.4.

The method **`doWork()`** in the class **`Goal`** (Figure 17.6) runs down the list

```
public class TD {
    static Input           input;
    static RuleStack       ruleStack;
    static DottedGoalStack dottedGoalStack;
    static KnownRuleGoals  knownRuleGoals;
    private static int     countDerivations;
    public static void parse(Input input) {
        TD.input = input;
        ruleStack = new RuleStack();
        dottedGoalStack = new DottedGoalStack();
        knownRuleGoals = new KnownRuleGoals();
        countDerivations = 0;
        System.out.println("Parsing \"" + input
                + "\" of length " + input.length());
        (new Goal(Grammar.startSym, 0, input.length())).doWork();
        System.out.println(countDerivations + " derivation"
                + (countDerivations == 1 ? "" : "s")
                + " found for string \"" + input + "\"\n");
    }
    public static void parsingFound() {
        countDerivations++;
        System.out.println("Parsing found:\n" + ruleStack.toString());
    }
}
```

Fig. 17.5. The class **TD**

```
public class Goal {
    Symbol lhs; int pos; int length;
    public void doWork() {
        for (int n = 0; n < Grammar.size(); n++) {
            Rule r = Grammar.getRule(n);
            if (r.lhs.equals(lhs))
                (new RuleGoal(this, r)).doWork();
        }
    }
}
```

Fig. 17.6. The class **Goal**

of grammar rules and for each rule **r** for the desired non-terminal it creates a **RuleGoal** containing the **Goal** and **r**, and activates it.

The method **doWork()** in the class **RuleGoal** (Figure 17.7) contains three calls to methods in **TD.knownRuleGoals**, which we ig-

```
public class RuleGoal extends Goal {
    Rule rule;
    public void doWork() {
        // avoid left recursion:
        if (TD.dottedGoalStack.contains(this)) return;
        // try to avoid rebuilding known parses:
        if (TD.knownRuleGoals.knownRuleGoalTable.containsKey(this)) {
            TD.knownRuleGoals.doWork(this); return;
        }
        TD.knownRuleGoals.startNewParsing(this);
        System.out.println("Trying rule goal " + toString());
        (new DottedGoal(this, 0, 0)).doWork();
    }
    public void doWorkAfterDone() {
        if (TD.dottedGoalStack.empty()) TD.parsingFound();
        else TD.dottedGoalStack.top().doWorkAfterMatch(length);
    }
}
```

Fig. 17.7. The class **RuleGoal**

nore until Section 17.3.4. For the moment we also ignore the call to **TD.dottedGoalStack.contains(this)**, which serves as a protection against problems with left recursion; it will be discussed in Section 17.3.3. So there is only one thing left to do for **RuleGoal.doWork()**: create a **DottedGoal** with both dots at the left end, and activate it. The method **doWorkAfterDone()** will be discussed later on.

The class **DottedGoal** (Figure 17.8) is the second-most complicated class of the parser, after the still mysterious class **KnownRuleGoals**. Its **doWork()** method is the first to show real action: it puts itself on the parsing stack **TD.dottedGoalStack**, puts the rule it contains on the rule stack **TD.ruleStack**, and leaves the upcoming intricate decisions to the private method **doAsTopOfStack()**.

This method has to distinguish between several situations. If the right-hand side of the rule is exhausted, there are two possibilities: the input segment is also exhausted, in which case the left-hand side of the dotted goal has been fully recognized; or it is not, in which case the dotted goal has failed and nothing needs to be done. If the left-hand side in this dotted goal has been fully recognized, we first signal this fact to **TD.knownRuleGoals** to be remembered for future use. Next we temporarily pop it off the dotted-goal stack. Since the fields **rhsUsed** and **inputUsed** are

```
public class DottedGoal extends RuleGoal {
    private int rhsUsed, inputUsed; // positions of dot in rhs and input
    public void doWork() {
        TD.dottedGoalStack.push(this);
        TD.ruleStack.push(rule);
        doAsTopOfStack();
        TD.ruleStack.pop();
        TD.dottedGoalStack.pop();
    }
    public void doWorkAfterMatch(int matchedLength) {
        // advance the dotted goal over matched non-terminal and input
        rhsUsed += 1; inputUsed += matchedLength;
        doAsTopOfStack();
        // retract the dotted goal
        rhsUsed -= 1; inputUsed -= matchedLength;
    }
    private void doAsTopOfStack() { // 'this' is top of parsing stack
        int activePos = pos + inputUsed;
        int leftoverLength = length - inputUsed;
        if (rule.rhs.length == rhsUsed) { // rule exhausted
            if (leftoverLength == 0) {    // input exhausted
                TD.knownRuleGoals.recordParsing(this);
                TD.dottedGoalStack.pop();
                ((RuleGoal)this).doWorkAfterDone();
                TD.dottedGoalStack.push(this);
            }
        } else {
            Symbol rhsAtDot = rule.rhs[rhsUsed];
            if (leftoverLength > 0) {
                Symbol inputAtDot = TD.input.symbolAt(activePos);
                if (rhsAtDot.equals(inputAtDot)) doWorkAfterMatch(1);
            }
            for (int len = 0; len <= leftoverLength; len++)
                (new Goal(rhsAtDot, activePos, len)).doWork();
        }
    }
}
```

Fig. 17.8. The class **DottedGoal**

now meaningless, the dotted goal reverts to being a rule goal, and a recognized one at that, and we continue our search by applying **doWorkAfterDone()** to it. (The qualifier **((RuleGoal)this)** in front of **doWorkAfterDone()** is actually superfluous, but serves to emphasize that we are now back dealing with a rule goal.) When done, we restore the old situation by pushing the present dotted goal back on the stack.

This method `RuleGoal.doWorkAfterDone()` (page 557) checks if the stack is empty. If it is, the start symbol, which was the `lhs` of the goal that was the first to be stacked, has been matched to the entire input, so a parsing has been found. If it is not, `doWorkAfterDone()` obtains the dotted goal on top of the stack, and calls its `doWorkAfterMatch(int matchedLength)` method. This method bumps the top of the stack over the matched length, and continues the search by calling `doAsTopOfStack()`, as before.

In this way both the flow of control and our explanation return to `DottedGoal`, where we were discussing `doAsTopOfStack()`. If we enter the `else` branch of the if-statement, the right-hand side of the rule is not exhausted, and there is a symbol `rhsDot` after the dot in it. Now there are several possibilities. The symbol `rhsDot` may be a terminal, in which case we want to check it against the input symbol. We first check if there is more input, and then compare it to the input symbol at `activePos`. If they match we call `doWorkAfterMatch(1)` to bump the top of the stack over one token, and continue searching; if they do not the goal has failed. If `rhsDot` is a non-terminal, it is tried against increasingly longer chunks of what remains of the input segment, by creating a new `Goal` for each chunk. This brings us back to the level of `Goal`s, and a new cycle in the top-down search can start.

Since the presented parser does not distinguish between terminals and non-terminals, we have a problem here: we cannot know if we should try the `if (rhsAtDot···` statement (for a terminal) or the `for (int len = 0; len <=···` statement (for a non-terminal). But since they are part of a search process we can solve this by just trying them both. One consequence of this is that we cannot prevent new `Goal`s from being created for a terminal symbol; since there is no syntax rule for a terminal, the for-loop in `Goal.doWork()` will find no match. Another consequence is that the program can also handle input in which some non-terminal productions have already been recognized.

We see that the methods `doWork()`, `doWorkAfterMatch(int)`, and the if-branch in `doAsTopOfStack()` are similar in structure: they all start with a modification of the situation, perform a recursive search, and then carefully restore the original situation when the recursive search returns. This technique allows all parsings to be found.

Note that besides the parse stack and the rule stack, there is also a search stack. Whereas the first two are explicit, the third is implicit and is contained in the Java recursion stack.

17.3.3 Handling Left Recursion

As explained in Section 6.3.1, a top-down parser will loop on a left-recursive grammar and the problem can be avoided by making sure that no new goal is accepted when that same goal is already being pursued. This is achieved by the test `TD.dottedGoalStack.contains(this)` in `RuleGoal.doWork()`. When a new goal is about to be put on the parse stack, `RuleGoal.doWork()`

tests if the **DottedGoalStack** already contains the goal **this**. If it does, the goal is not tried for the second time, and **RuleGoal.doWork()** returns immediately.

The above program was optimized for brevity and, hopefully, for clarity. With an empty implementation of the methods in the class **KnownRuleGoals**, however, it may require an amount of time that is exponential in the length of the input. The optimization in the following section remedies that.

17.3.4 Parsing in Polynomial Time

An effective and relatively simple way to avoid exponential time requirement in a context-free parser is to equip it with a *well-formed substring table*, often abbreviated to *WFST*. A WFST is a table which holds all partial parse trees for each substring (segment) of the input string; it is very similar to the table generated by the CYK algorithm. It is can be shown that the amount of work needed to construct the table cannot exceed $O(n^{k+1})$ where n is the length of the input string and k is the maximum number of non-terminals in any right-hand side. This takes the exponential sting out of the depth-first search.

The WFST can be constructed in advance (which is what the CYK algorithm does), or while parsing proceeds ("on the fly"). We shall do the latter here. Also, rather than using a WFST as defined above, we shall use a *known-parsing table*, which holds the partial parse trees for each **RuleGoal**, i.e., each combination of a grammar rule and a substring of the input. These two design decisions have to do with the order in which the relevant information becomes available in the parser described above.

An implementation of the known-parsing table is shown in Figure 17.9. The class **KnownRuleGoals** supplies the methods **startNewParsing(RuleGoal)**, **recordParsing(RuleGoal)** and **doWork(RuleGoal)**.

The parser interacts with the known-parsing table **TD.knownRuleGoals** only in a few places. The first place is in **RuleGoal.doWork()**, where a call is made to **knownRuleGoalTable.containsKey(this)**. This method accesses the known-parsing table to find out if the rule goal **this** has been pursued before. When called for the very first time, it will yield **false** since there are no known parsings yet. So we skip the statements controlled by the if, and land at the call of **startNewParsing(this)**. This prepares the table for the recording of the zero or more parsings that will be found for **this**. Once the parsings have been found and the test is made a second time, it succeeds, and, rather than trying the rule goal again, a call to **TD.knownRuleGoals.doWork(this)** is made, which produces the zero or more parsings from the known-parsing table. The last interaction between the parser and the known-parsing table is in **DottedGoal.doAsTopOfStack()**, where a call of **recordParsing()** is used to enter the discovered parsing.

The rule goals are recorded in a three-level data structure. The first level is the hash table **knownRuleGoalTable**, indexed by **RuleGoal**s; its elements are objects of class **KnownRuleGoal**. A **KnownRuleGoal** has **ruleGoal** field and a **knownParsingSet** field, which is a vector of objects of class **KnownParsing** and which forms the second level. Each **KnownParsingSet** contains a parse tree

```
import java.util.ArrayList;
import java.util.Hashtable;
public class KnownRuleGoals {
  Hashtable<RuleGoal,KnownRuleGoal> knownRuleGoalTable
      = new Hashtable<RuleGoal,KnownRuleGoal>();
  Hashtable<RuleGoal, Integer> startParsingTable
      = new Hashtable<RuleGoal, Integer>();
  public void startNewParsing(RuleGoal ruleGoal) {
    startParsingTable.put(ruleGoal, new Integer(TD.ruleStack.size()));
    knownRuleGoalTable.put(ruleGoal, new KnownRuleGoal(ruleGoal));
  }
  public void recordParsing(RuleGoal ruleGoal) {
    knownRuleGoalTable.get(ruleGoal).record();
  }
  public void doWork(RuleGoal ruleGoal) {
    knownRuleGoalTable.get(ruleGoal).doWork();
  }
  private class KnownRuleGoal {
    RuleGoal ruleGoal;
    ArrayList<KnownParsing> knownParsingSet =
        new ArrayList<KnownParsing>();
    void record() {
      knownParsingSet.add(new KnownParsing());
    }
    void doWork() {
      for (int i = 0; i < knownParsingSet.size(); i++) {
        knownParsingSet.get(i).doWork();
      }
    }
    private class KnownParsing {
      Rule [] knownParsing;
      KnownParsing() {
        int stackSizeAtStart =
          startParsingTable.get(ruleGoal).intValue();
        int stackSize = TD.ruleStack.size();
        knownParsing = new Rule[stackSize - stackSizeAtStart];
        for (int i = stackSizeAtStart, j = 0; i < stackSize; i++, j++) {
          knownParsing[j] = TD.ruleStack.elementAt(i);
        }
      }
      void doWork() {
        int oldStackSize = TD.ruleStack.size();
        for (int i = 0; i < knownParsing.length; i++) {
          TD.ruleStack.push(knownParsing[i]);
        }
        ruleGoal.doWorkAfterDone();
        TD.ruleStack.setSize(oldStackSize); // pop all
      }
    }
  }
}
```

Fig. 17.9. Implementation of the known-parsing table

for the described goal, represented as an array of **Rule**s called **knownParsing**; this is the third level.

There is also a hash table **startParsingTable**, indexed with **RuleGoal**s, which holds the level of the rule stack at the time when work on the **RuleGoal** was started.

With this knowledge of the data structure we can easily understand the methods in **KnownRuleGoals**. The method **startNewParsing(RuleGoal)** records the size of the rule stack in the hash table **startParsingTable**, with the rule goal as an index. The idea is that when a complete parsing is found for the rule goal, the segment of the rule stack between the recorded size and the actual size contains that parsing. It also puts an empty **KnownRuleGoal** in the known-parsing table; if no parsings are found for the rule goal, this empty entry serves to record that fact.

When the main mechanism of the parser has found a parsing for the rule goal (in **DottedGoal.doAsTopOfStack()**) it calls **recordParsing()**. This method is very simple: it retrieves the **KnownRuleGoal** for the rule goal, and applies its **record()** to it.

The method **record()** in **KnownRuleGoal** creates a new **KnownParsing** and adds it to the **knownParsingSet**. The constructor for the **KnownParsing** retrieves from the hash table **startParsingTable** the rule stack size at the time the parsing for this rule goal started and the present rule stack size, creates the array **knownParsing[]** and copies the parsing into it from the segment of the rule stack between the present stack size and **stackSizeAtStart**. The parsing is now safely stored in the array in the vector in the hash table.

Note that as long as the rule goal is under construction, it is also on the parse stack. This means that the left-recursion protection test **TD.dottedGoalStack.contains(this)** in **RuleGoal** cannot yield **false** until after the rule goal has been rebmoved from the parse stack, i.e., after all its parsings have been found. So when **TD.knownRuleGoals.doWork(this)** is called eventually, we can be sure that it can reproduce all possible parsings.

This brings us to the method **doWork(RuleGoal)** in **KnownRuleGoals**; it retrieves the **KnownRuleGoal** for the rule goal, and applies its **doWork()** to it. This method steps through the **knownParsingSet** and applies **doWork()** to each **KnownParsing** in it. The method **KnownParsing.doWork()** is the first in the known-parsing class that directly contributes to the parsing activity. It copies the parsing stored in **knownParsing** to the rule stack as if it had been performed normally, signals to the rule goal that the rule goal has been recognized, and removes the copied parsing. The net result of the call to **TD.knownRuleGoals.doWork(this)** is that **RuleGoal.doWorkAfterDone()** is called exactly as many times and in the same situations as if the rule goal were parsed without the use of the known-parsing table, only much faster.

It will be obvious that copying a ready-made solution is much more efficient than reconstructing that solution. That it makes the difference between exponential and polynomial behavior is less obvious, but true nevertheless. When applied to the input strings **())** and **(((()))))**, the parser without the known-parsing table tries 41624

rules for the `UserGrammar4` example, the parser with it only 203. A parser that does not use the known-parsing table can be obtained simply by putting `false &&` in front of the test of `containsKey` in `RuleGoal.doWork()` (page 557).

17.4 Programming Language Paradigms

A programming paradigm is a mind set for formulating and solving programming problems. A paradigm is characterized by a single principle, a finite set of concepts to support the principle, an infinite set of methods to apply the concepts to solve the problem, and, hopefully, a user-community-cum-culture, with books, user groups, etc., to spread the word. There are four major programming paradigms: imperative ("do this, then do that"); object-oriented ("everything is an object, with buttons (methods) on the outside"); functional ("everything is a function from input to output"); and logic ("everything is a set of relations held together by Horn clauses").

There is also parallel and distributed programming, but that is rather a — hopefully beneficial — restriction than a programming paradigm, in that it affects the nature of the algorithms expressible in it. It is covered in Chapter 14.

Although in principle anything programmable can be programmed in any paradigm, some combinations are much more convenient than others, and it is interesting to see how the different paradigms relate to the various programs arising in parser writing. Figures 17.1 and 17.2 show the four kinds of programs that are likely to be produced by a parser writer: the interpreting parser, the parser generator, the generated table-driven parser, and the compiled parser.

Interpreters and parser generators are just programs, no different in their nature than any other programs; they can be written in any language in any paradigm the programmer finds convenient. An example of an interpreter in Java was given in Section 17.3.

Table-driven parsers do not contain much in the way of programming: just a simple loop accessing a table. The imperative paradigm is no doubt the best for this; there are no obvious objects, and functional and logic languages are not very good at handling large matrices. On the other hand, table-driven parsers often contain semantic routines, and these may dictate the programming language and the paradigm.

We now turn to compiled parsers, parsers for which code in some programming language must be generated. One powerful method for creating parser code is to generate a parsing routine for each non-terminal, as in recursive descent (Sections 6.6 and 8.2.6). The idea was first suggested in the beginning of 1961 by Grau [332], but he could not implement the idea because he had no compiler capable of handling recursive routines. The first explicit description is by Lucas [41] later that year. The idea was formalized by Knuth [43] in 1971. In the next few sections we discuss the generation of recursive descent parsers in the four major paradigms.

17.4.1 Imperative and Object-Oriented Programming

Constructing a deterministic (LL(1)) parser by compiled recursive descent is very simple in an imperative language, once the look-ahead sets have been computed;

Section 8.2 describes the technique. Doing the same in an object-oriented language while respecting the object-oriented paradigm is much harder. The reason is that one would like to encapsule all information about a non-terminal *A* in a single object, **parse_***A*, but the problem is that the look-ahead sets of the alternatives of *A* belong to *A* but depend on many other non-terminals. One can of course program around this but the resulting code is awkward.

Two possible solutions to this problem have been suggested. The first is by Koskimies [342], who lets each **parse_***A* start with empty look-ahead sets. The first time *A* is called and has to decide between its alternatives, it calls them one by one. The routines for the alternatives then report back their look-ahead sets, and *A* assembles its from theirs. The second is by Metsker [357], who hides the details in a toolkit which defines the classes **Repetition**, **Sequence**, **Alternation** and **Word**. A parser can then be constructed in object-oriented fashion from these.

17.4.2 Functional Programming

Although functional languages are less than great for deterministic, table-driven parsers (but see Leermakers et al. [346]), they are very convenient for backtracking recursive descent parsers. The three mechanisms in a CF grammar are concatenation, choice and identification, as explained on page 24. The idea is to have the first two of these mechanisms as higher-order functions in our parser; the identification comes free of charge through the programming language.

A higher-order function is a function that takes functions as its parameters; in our parser these parameters will be functions that parse given non-terminals, and the two higher-order functions, called *combinators*, coordinate the parsing. *Combinator parsing* was first described by Frost [344] and Hutton [345], but we shall follow here Frost and Szydlowski [353], who give the following short, self-contained example.

The code for a parser for the grammar **S→aSS**|ε in the functional language Haskell using combinators is

```
s = (a `and_then` s `and_then` s) `or_also` empty
a = term 'a'
```

We see that the translation is immediate (with concatenation represented by **`and_then`** and choice by **`or_also`**), but it does require a lot of explanation.

The Haskell function for parsing a non-terminal *A* accepts one parameter as input, a list of strings. It picks up each string from the list in succession and tries to find one or more prefixes in it that correspond to *A*. If it finds any, it adds the strings that remain after the prefixes have been removed to the output list; otherwise it adds nothing to the list.

Suppose *A* produces **[a|b]*b** and we call its parsing function with a list of three strings **["aab","aa","abbaba"]**, then the output is a list of four strings: **["","a","aba","baba"]**. The first, empty string results from **"aab"**, from which the full **aab** has been removed as a prefix; the second input string has no prefix that matches *A*, so it does not feature in the output; and the last three strings result

from **"abbaba"**, from which **abbab**, **abb**, and **ab** have been removed as prefixes successively. (Actually the Haskell system produces the strings in a different order, due to its particular search order.)

There are two things to be noted here. The first is that a completed parsing results in an empty string to be appended to the list of strings, and vice versa an empty string indicated a successful parsing; this is the way we shall interpret the final result. The second is that the output list of strings may be shorter or longer or equally long as the input list, depending on how many parsings fail, deleting strings, and how many are locally ambiguous, producing more than one string. The strings themselves can only get shorter, or keep the same length. So, although the routine for *A* seems to act as a filter, letting only those strings pass whose prefixes match *A*, it differs from a filter in that it can duplicate the things passing through it.

Now we must implement the combinators **`and_then`** and **`or_also`**; the backquotes around the names indicate to the Haskell system that they are infix functions. The other two functions, **term** and **empty**, will be defined later. The **`or_also`** combinator is simple:

```
(p `or_also` q) inputs = (p inputs) ++ (q inputs)
```

which says that the function **(p `or_also` q)** is applied to the parameter **inputs** by applying the function **p** to it, then applying **q** to it, and concatenating (**++**) the two resulting lists. Note that the input list gets copied here: both **p** and **q** start with the same lists of inputs and their contributions are combined.

The combinator **`and_then`** is more complicated (see Problem 17.5):

```
(p `and_then` q) inputs | (r == []) = []
                        | (r /= []) = (q r)
                          where
                              r = (p inputs)
```

It features a local variable **r**, which is computed first, by applying **p** to the list of strings. So **r** is the list of strings that remain after a prefix matching **p** has been removed from them. If that list is empty (that is, no string had a prefix matching **p**), we return the empty list; if it is not, we apply **q** to it and return the result. Any string in that result corresponds to a string in **inputs** that had prefixes matching **p** followed by **q** removed from it.

Remarkably, the function that parses a terminal symbol (that is, removes it as a prefix) looks even more forbidding:

```
term c []     = []
term c (s:ts) = (term_c s) ++ (term c ts)
                where
                    term_c ""                = []
                    term_c (c1:s) | (c1 == c) = [s]
                                  | (c1 /= c) = []
```

but that is because it has to take the list apart and then the strings in it, to get at the first tokens of these strings, and then reassemble the lot. The first definition of **term c inputs** says that if the input is the empty list, so is the output. The second

says that if the input can be split into a string **s** and the rest of the strings **ts**, then the output is composed by applying an auxiliary function **term_c** to the string **s** and concatenating the result with the result of the original function working on the rest of the strings, **ts**.

The definition of the function **term_c** follows in the **where** section. The first line says that if its parameter is the empty string, the result is the empty list. The next two lines test for the first token of the parameter: if it is **c**, the token we want to match, we return the rest of the string packed in a list, otherwise we return the empty list.

Note how both the empty string and the non-matching string are turned into the empty list. The empty list is then concatenated with the rest of the list, which makes it disappear from the game: the empty list indicates failure. On the other hand, a call of **term c s** where **s** is **"c"** results in a list of one element, the empty string, **[""]**. This empty string is just a new element of the output list; it stays in the game, and can in the end signal success. "Emptiness" is a subtle thing...

The function representing ε is very simple:

```
empty inputs = inputs
```

It just copies the input list.

We now have a complete program, and can call the parsing routine for **S** on an input string, say **aaa**. To do this, we write

```
s ["aaa"]
```

and the system responds with

```
["","","","a","","","a","aa","aaa"]
```

This reports five successful parsings and four failures. This technique can handle any non-left-recursive CF grammar.

Some of the above functions can be written much more compactly and elegantly in Haskell, but we have chosen the forms shown here because they require the least knowledge of Haskell.

The above describes the bare bones of functional parsing, and many additions and improvements need to be made. We will mention a few here, but recent literature supplies many more. A filter is needed to clean out failed attempts. A semantics mechanism must be added, since the "parser" we constructed above is actually a recognizer; Hutton [345] describes how to do that. The naive parser has exponential time complexity, which can be reduced to cubic by memoization; see for example Frost [350] or Johnson [351]. Even more efficiency can be gained by doing partial evaluation, as explained by Sperber and Thiemann [356].

Ljunglöf [358] describes extensively how to program parsers in Haskell, and includes detailed code for Earley and chart parsers. Several books on functional programming have a section on parser writing, for example Thompson [413, Sect. 17.5].

The functional paradigm has also made great contributions to the field of natural language parsing, in the form of the many natural language parsers written in Lisp.

17.4.3 Logic Programming

If functional languages are a good vehicle for compiled parsers, logic languages, more in particular Prolog, are an even better fit. The examples in this book (Definite Clause Grammars in Section 6.7, Cancellation Parsing in Section 6.8, and parsing VW grammars in Section 15.2.3), and the many papers listed in the sections on non-Chomsky systems ((Web)Section 18.2.6), natural language parsing (18.3.5), parser writing (18.3.1), and parsing as deduction (18.3.4), are ample proof of that. Prolog also plays a considerable role in natural language parsing.

Prolog has two major advantages over most other paradigms: the search mechanism is built-in, handling the context-free part; and the semantics can be manipulated conveniently with logic variables, handling the context-sensitive part. Logic variables are subject to unification, a very powerful data-manipulation mechanism not easily programmed in the other paradigms.

Only the constraint programming paradigm is more powerful, but it is experimental, and no complete implementation technique for it is known or perhaps even possible. Still, progress is being made on it, and it finds its way into parsing; see, for example, Morawietz [373] or Erk and Kruijff [374].

17.5 Alternative Uses of Parsing

Parsing is the structuring of text according to a grammar, no more, no less. In that sense there cannot be alternative uses of parsing. Still, some applications of parsing are unusual and perhaps surprising. We will cover three of them here: data compression, machine instruction generation in compilers, and support of logic languages. In the first two, text structuring is still prominent, but the third may qualify as "alternative use of parsing techniques".

17.5.1 Data Compression

Files can be compressed only if they contain redundancy, but most files people use have some. Usually this redundancy is internal: if we find that a file contains the word **`aardvark`** many times, we can replace it by **`@23`** provided **`@23`** does not occur otherwise. This is the kind of redundancy that is exploited by various *zip* programs. But redundancy can also be external: if both the sender and the receiver know that a file contains a tax declaration, only the numbers with the names of the boxes they go into have to be stored in the file, not all the surrounding text. This kind of redundancy is exploited by data bases. In both cases the important point is that we know something about the file, either by inspection or as advance knowledge.

Knowing that a file conforms to a CF grammar amounts to a lot of information, and over the last decade progress has been made to utilize that knowledge. Java applets that have to be sent over the Internet have been especially interesting targets for the technique. Such programs are transmitted in source or byte code, so they can

be subjected to user security checks before they are run on the receiver's computer (Evans [366]).

The idea is to store the sequence of rule numbers that produced the file — its leftmost derivation, see Section 3.1.3 — rather than the file itself, in what is called *grammar-based compression*. In terms of Java this means that, given that rule 75 of the grammar is

```
ForStatement  →  for ( ForInitOption ; ExpressionOption ;
                 ForUpdateOption ) Statement()
```

it is cheaper to store **75,1** for non-terminal 75, alternative 1, than the produced text **for (... ; ... ; ...)**. The **75,1** can be stored in 11 bits: 7 for the non-terminal number since there are fewer than 128 non-terminals in the Java grammar, and 4 bits for the alternative number since there are at most 16 alternatives to any non-terminal. The produced text form would cost 7 bytes, = 56 bits, so in this case storing the leftmost derivation saves 45 bits, which is 80.4%.

Actually it is better than that: leftmost derivation implies leftmost production at the receivers end, and in leftmost production we know at any moment which non-terminal we are going to expand: the leftmost one in the sentential form. So we do not have to store the non-terminal number at all, and 4 bits for the alternative number suffice, raising our savings to 52 bits, 92.9%. Even better, since the non-terminal **ForStatement** has only one alternative, we do not need to store the alternative number at all, bringing us an untoppable savings of 100%! Of course, in a sense this is cheating since the information that the next non-terminal is **ForStatement** was supplied by the preceding **Statement**, at a cost of 4 bits since it has 16 alternatives. Still, savings can be considerable, as we will see.

This juggling of alternative numbers, bits, and compression rates soon gets messy and a more systematic approach is needed. Also, we want to compare the performance of grammar-based compression to the standard Lempel–Ziv compression.

As a highly simplified example we will use files conforming to the grammar

```
1.  S_S  →  a S a
2.       |  b S b
3.       |  c
```

We start with a random file obeying the above grammar, 1 000 001 bytes long, and starting with **bababba...**. Since the file contains only three different tokens, and one of them only once, we expect the file to compress well under traditional techniques and indeed *gzip* reduces its size to 159 107 bytes (84.1%) (see the table in Figure 17.10).

The compressed version is now constructed as follows. LL(1) parsing of the file immediately reveals that the top-most rule of the parse tree is **S→bSb**, which is rule number 2. There are three rules to **S**, so specifying the alternative will require 2 bits. Since there is no rule 0, the three rules can be specified by the 2-bit integers 0, 1, and 2. So for the first byte of the file, the two bits 01 are output. The next byte is an **a**, which is produced by rule 1, so 00 is appended. The next two bytes yield 0100, which completes one byte of the compressed file: 01000100. This process is repeated until after 500 000 bytes we reach the **c**; in the meantime we have output 500 000/4

Object	Size in bytes	Compression rate
Demo file	1 000 001	0%
Demo file, zipped	159 107	84.1%
With naive parsing	125 001	87.5%
With naive parsing, zipped	71 909	92.8%
Parsed with 1 expanded rule	93 793	90.6%
Parsed with 1 expanded rule, zipped	67 496	93.3%
Parsed with 253 expanded rules	62 684	93.7%
Parsed with 253 expanded rules, zipped	62 717	93.7%

Fig. 17.10. The effects of various stages of grammar-based compression

= 125 000 bytes in the compressed file. The `c` is parsed by rule 3, so we output the bits 10. Now the second half of the input file has already been completely predicted by the LL(1) parser, so no more rule numbers are produced. We fill up the last byte with arbitrary bits, and we are done. The resulting size is 125 001 bytes (87.5%), which is already better than *gzip* did.

The receiving program starts with `S` as the initial form of the reconstructed file. It then reads the first byte of the compressed file, 01000100, extracts the first two bits, concludes that it needs to apply rule number 2 of non-terminal `S`, and replaces the reconstructed form by `bSb`. The next 2 bits turn it into `baSab`, etc. When the code 10 is found, it identifies rule number 3, and the `S` gets replaced by `c`. Now there is no non-terminal left in the reconstructed form, so the process stops, the last few bits in the input are ignored and the reconstructed form is written to file.

Since Lempel–Ziv compression is completely different from grammar-based compression, it is tempting to apply it to the compressed file. Indeed the size reduces further, to 71 909 bytes, and it is easy to see why. The two-bit integers we were writing to the compressed file can only have the values 00, 01, and 10, and the last value occurs only once; so we are using only 50+ε% of the capacity. This suggests that we would have done better with a rule with 4 alternatives rather than 3. That can be arranged, by substituting one non-terminal A in the right-hand side of some rule B by a right-hand side of A. We can, for example substitute `S→aSa` into `S→bSb`, resulting in `S→baSab`; this will be our rule number 4. The results are shown in the third section of Figure 17.10, and we see that it helps considerably; but additional zipping still works, so there is still redundancy left. (Another way to remedy the bad fit is by using adaptive arithmetic coding, as reported by Evans [366].)

If four alternatives are better than three, more might be even better; and it is. Evans [367] shows that it is efficient to substitute out non-terminals until they all have 256 alternatives. Then each alternative number fits in exactly one byte, which speeds up processing in both the compressing and the decompressing side, and no space is lost. This immediately raises the question which non-terminals to substitute in which

right-hand sides. Evans gives heuristics, possibly involving analyzing the input file in advance, but since we want to keep it simple in our example, we substitute the rules **S→aSa** and **S→bSb** in each other until we have rules that start with all combinations of 8 **a**s and **b**s. That yields 256 rules, and because we still need our first three rules, we just discard the last three rules. This means that our grammar now looks as follows:

```
1.    Ss  →  a S a
2.           |  b S b
3.           |  c
4.           |  a a a a a a a a S a a a a a a a a
5.           |  a a a a a a a b S b a a a a a a a
                ...
255.         |  b b b b b a b b S b b a b b b b b
256.         |  b b b b b b a a S a a b b b b b b
```

When the input starts with one of the discarded combinations, for example **bbbbbbab**, or when the **c** is among them, rules 1 or 2 take over. Using this grammar reduces the size to 62 684 bytes (93.7%), which is very close to the theoretical minimum of 62 501 (1 bit for each **a** or **b** in the first half, + 1 bit for the **c**). It is gratifying to see that we have finally reached a compression that cannot be improved by an additional application of *gzip*.

In the above explanation we have swept an important problem under the rug: after the substitutions the grammar is no longer LL(1); it is even ambiguous. There are several ways to solve this. We observe that the above grammar is LL(8), provided dynamic conflict resolvers are attached to rules 1 and 2 to avoid these rules when possible. It is not inconceivable that such an adaptation can be automated; see Problem 17.6. Evans [367] shows how to rig an Earley parser so it always recognizes the longest possible sequence. And even in the absence of such solutions, spending considerable time compressing a file is worth while, when the result is used sufficiently often.

Two notes: Many papers on data compression use the term "parsing" in the sense of repeated string recognition, and as such these techniques do not qualify as "applications of parsing". And for readers who read Russian, some papers on grammar-based data compression are in Russian, for example Kurapova and Ryabko [352].

17.5.2 Machine Code Generation

A large part of program code in imperative languages consists of arithmetic expressions. The compiler analyses these expressions and makes all implicit actions in them explicit; examples are indirection, subscripting, and field selection. These explicit expressions, which are part of the intermediate code (IC) in the compiler, very quickly get more complicated than one would expect. For example, the integer expression **a[b]** is converted to something like **M[a+4×b]**: the value of **b** must be multiplied by 4 since integers are 4 bytes long, the address of the array **a** must be added to it, and the memory location at that address must be read.

In the end such intermediate code expressions must be converted to machine instructions, which can also be seen as expressions. For example, most machines have an *add constant* instruction, which adds a constant to a machine register. It could be written **ADDC** R_n**,c** and be represented by the expression R_n**:=c+**R_n.

One way to translate from intermediate code to machine code is to generate the IC expressions according to an IC grammar and to reparse this stream according to a machine code grammar. That is exactly what Glanville and Graham [336] do; they use a modified SLR(1) parser for the process. The technique is variously referred to as Glanville–Graham and Graham–Glanville, which again goes to show that techniques should not be named after people. We will call it *expression-rewriting code generation*, in line with the better known tree-rewriting code generation.

Intermediate and machine code grammars are large and repetitive: the SLR(1) parser for the very simple example in Glanville and Graham's paper already has 42 states. We shall therefore give here a totally unrealistic example and just sketch the process; for more details we refer to the above paper and to literature on the Internet.

Suppose we have a machine with the following six machine instructions:

Name	Rule	Assembler	Cost
Add constant **c**	**R[n] → + c R[n]**	**ADDC** R_n**,c**	1
Multiply by $-128 \leq$ **c** ≤ 127	**R[n] → × c R[n]**	**MULSC** R_n**,c**	2
Multiply by constant **c**	**R[n] → × c R[n]**	**MULC** R_n**,c**	3
Load address of variable **v**	**R[n] →** A_v	**LA** R_n**,v**	1
Load address of an element of array **a** (**c**$_2$=4)	**R[n] → +** A_a **× c R[i]**	**LAAE** R_n**,a,[**R_i**]**	3
Load value from memory	**R[n] → @ R[i]**	**LD** R_n**,M[**R_i**]**	3

The expressions are presented in prefix form: the R_n**:=c+**R_n above shows up as **R[n] →+cR[n]**. There are two instructions for multiplication, one with a small (one-byte) constant, and another with any constant. The **LAAE** instruction Loads the Address of an Array Element. The **LD** instruction loads one register with the value of the memory location at (**@**) another register. The column marked "Assembler" shows the machine instructions to be generated for each rule. "Cost" specifies the cost in arbitrary units. The grammar contains two kinds of context or semantic conditions. The first is that the register numbers must be substituted consistently: the first line is actually an abbreviation for

Add constant	**R[1] → + c R[1]**	**ADDC** R_1**,c**	1
Add constant	**R[2] → + c R[2]**	**ADDC** R_2**,c**	1
...			

The second is the condition $-128 \leq$ **c** ≤ 127 on the **MULSC** instruction and the **c**$_2$=4 in **LAAE**.

The grammar is ambiguous and certainly not SLR(1). There are many ways to resolve the conflicts; we will resolve shift-reduce conflicts by shifting and re-

duce/reduce conflicts by reducing with the longest reduce with the lowest cost which is compatible with the semantic restrictions.

We now return to our source code expression **a[b]**, which translates into the intermediate code expression **@+A**$_a$×**4@A**$_b$. Here **A**$_a$ is a constant, equal to the machine address of the first word of the array **a**, and **A**$_b$ is the address of the variable **b** in memory. When this expression is parsed, the parser goes through a number of shift/reduce conflicts, and shifts it completely onto the stack.

The further actions are shown in Figure 17.11. Initially only one reduction can

Stack	Action	Machine instruction	Cost
@ + A$_a$ × **4 @ A**$_b$	reduce, Load addr. of var.	**LA R**$_1$**,b**	**1**
@ + A$_a$ × **4 @ R**$_1$	reduce, Load value from mem.	**LD R**$_2$**,M[R**$_1$**]**	**3**
@ + A$_a$ × **4 R**$_2$	reduce, Load addr. of ar. elem.	**LAAE R**$_3$**,a[R**$_2$**]**	**3**
@ R$_3$	reduce, Load value from mem.	**LD R**$_4$**,M[R**$_3$**]**	**3**
R$_4$	result left in **R**$_4$		**10**

Fig. 17.11. Parser actions during expression-rewriting code generation

be done, using the rule for loading the address of a variable. It replaces the **A**$_b$ on the stack by **R**$_1$, while at the same time issuing the instruction **LA R**$_1$**,b**. We see that reducing the top segment T of the stack to a register R corresponds to code which at run time puts the value of the expression corresponding to T into R: the combined action leaves the semantics unaltered — the basic tenet of code generation.

The second reduction is also forced: load value (that of **b**) from memory; to simplify matters we use a very simple register allocation scheme here: just assign a new register every time. The next stack configuration, however, has a triple reduce/reduce conflict: it can be reduced with the instructions **MULSC**, **MULC**, and **LAAE**. The first matches because the constant (4) is small, the second has no restrictions and the third matches because the constant is in range. The criterion to take the cheapest of the longest reductions that fit the restrictions leads us to use **LAAE**. The last step is again forced, and leaves the result of the expression **a[b]** in register 4, at a total cost of 10 units.

Now suppose the source expression had been **&a+5**×**b**, where **&a** is the address of **a** in a C-like notation. This would have resulted in an intermediate code expression **+A**$_a$×**5@A**$_b$. The first two steps in the reduction sequence remain the same, but in the resulting stack configuration **+ A**$_a$ × **5 R**$_2$ the constant is not 4, and **LAAE** no longer fulfills the restrictions. The other two reductions do, however, and the cheapest is chosen:

+ A$_a$ × **5 R**$_2$	reduce, Multiply by small const.	**MULSC R**$_3$**,5**	**2**
+ A$_a$ **R**$_3$	reduce, Add constant	**ADDC R**$_3$**,a**	**1**
R$_3$	result left in **R**$_3$		**9**

We see that the algorithm automatically adapts to a small change in the intermediate code expression by generating quite different code. Combined with the possibility to impose context restrictions and assign costs, expression-rewriting code generation

is a versatile tool for generating good code for expressions. (More recently it has been superseded by BURS-techniques, which rewrite trees rather than expressions.)

There are several problems with the above approach; they are all concerned with the fact that we are using an ambiguous grammar to do parsing. So we run the risk of making a decision that will turn out to block progress further on. See Glanville and Graham's paper [336] for solutions.

17.5.3 Support of Logic Languages

In Sections 6.7 and 17.4.3 we have seen how a logic language, in this case DCG-extended Prolog, can be used to implement parsing. This works both ways: it is also possible to use parsing to support the inference process involved in logic languages. Normally logic languages use a built-in top-down depth-first search, as do many parsing algorithms, but many of the latter also incorporate some breadth-first and/or bottom-up component. The idea is to use the well-balanced search techniques from the parsing scene to guide the inference process in the logic language. This usually requires imposing some restrictions on the logic languages.

Rosenblueth [370] uses chart parsers as inference systems for logic programs in which the arguments of the logic predicates can be classified as either input or output, a restriction reminiscent of that on the attributes in attribute grammars. In [371] Rosenblueth and Peralta do the same using SLR parsing. Vilain [369] exploits tabular Earley parsing to implement deduction recognition in a frame language.

17.6 Conclusion

Broadly speaking, general CF parsing is best done with a GLR(0) parser, and the same goes for general CF substring parsing. For linear-time parsing strong-LL(1) and LALR(1) are still good choices. For linear-time substring parsing Bates and Lavie's technique [214] is the prime candidate.

Parsers can perform the semantics of their input immediately, or can create parse tree(s) to be processed further. In a parser, the grammar can be interpreted, compiled into a table or compiled into program code, in that order of efficiency.

Recursive-descent parsers can be implemented efficiently in all four paradigms, imperative, object-oriented, functional, and logic, with especially the latter having remarkable properties.

Parsing can be used outside the traditional setting of matching a string to a grammar; examples are data compression, machine code generation and the support of logic languages.

Problems

Problem 17.1: How does the parser from Section 17.3 handle infinite ambiguity?

Problem 17.2: Modify the parser from Section 17.3 so it delivers a parse forest rather than a sequence of parsings. This would allow infinite ambiguity to be represented better.

Problem 17.3: The explicit dotted-goal stack in the parser from Section 17.3 can be avoided by linking each dotted goal to its parent. Modify the code in that sense.

Problem 17.4: *History:* Determine the parsing technique used by Grau in his 1961 paper [332]. That is, design an algorithm that produces Grau's table or something close, making reasonable assumptions about the grammar used.

Problem 17.5: Why can we not just define `(p 'and_then' q) inputs` as `q (p inputs)`, which is exactly what `'and_then'` seems to mean?

Problem 17.6: *Project*: Given an LL or LR grammar, redesign the corresponding linear-time parser so it is still linear-time when rules are substituted, as described in Section 17.5.1.

Problem 17.7: *Project*: Expression-rewriting code generation uses ambiguous grammars and bottom-up parsing, and requires that no grammar-conforming input be rejected, regardless of how reduce/reduce conflicts are resolved. Glanville and Graham [336] give conditions on the grammar to achieve this, but these conditions, although adequate for their purpose, seem overly restrictive. Try to find more lenient conditions and still accept any correct input, under two regimes: 1. shift/reduce conflicts are always resolved by shifting; 2. shift/reduce conflicts can be resolved any way the code generator sees fit.