CS2 Language Processing note 10

Table-driven predictive parsers

The overall picture

In the last note we saw that top-down parsing can be done quite straightforwardly provided our grammar possesses certain good properties — in fact, provided it is what is called an *LL(1) grammar*. This is because for these grammars it is possible to draw up a *parse table* telling us which rule to apply in any given situation; once we have such a parse table, our method of top-down parsing can proceed completely automatically. We also saw that certain naturally arising grammars (such as the grammar with two kinds of **if** statements) are not LL(1) as they stand, and so are not immediately suitable for predictive parsing.

Typically, given a context-free grammar G, we can build a parser for the language $L(G)$ by carrying out the following three steps:

1. If G is not already an $LL(1)$ grammar, transform G into an equivalent grammar G' which *is* $LL(1)$, by eliminating problematic features such as ambiguity, left recursion, and *common prefixes* as in the **if** example.
2. Given an $LL(1)$ grammar G', calculate the parse table for G'.
3. Implement a simple algorithm for parsing a given sentence with the help of the parse table for G'.

In the rest of this thread, we will consider these three stages in more detail.

There is no fully automatic method for stage 1 — this stage typically requires human understanding of what the language is supposed to mean and how it is to be used. Nevertheless, there are some useful methods and general principles that can help, and we will be covering these in Note 11. Stage 2 is fairly easy to carry out by hand for small grammars, but for larger examples it is preferable to use an automatic method for calculating the parse table, and we will present such a method in Note 12. For the remainder of this note, we will describe stage 3 in more detail. The algorithm in question is essentially the method of top-down parsing that we have already described in Note 9, but here we will be a little more concrete about what parse tables are, and how a table-driven predictive parser may be implemented in an efficient way.

Notice that the algorithm for stage 3 is executed every time we use the parser to parse a sentence, whereas stages 1 and 2 are carried out “once and for all” (for the language in question) when we are building our parser. However, the *implementation* of stage 3 is obviously part of the parser construction phase.

What is more, the algorithm here is the same whatever the language — in order to configure it for a particular grammar, we just need to supply an appropriate parse table. In other words, it is possible to build a general *parsing engine* that can take a parse table for a grammar and use that to parse input sentences. This means that extra effort spent on making the parsing engine correct, efficient and robust is well repaid over time, since it can be reused for many different languages.

Parse tables

Recall the following grammar from Note 9 for the simple circuit description language.

$$C \longrightarrow \textbf{seq}\ B \mid \textbf{par}\ B \mid \mathsf{value} \qquad\qquad B \longrightarrow C\ B \mid \textbf{end}$$

Call this grammar G_1. We have already seen how G_1 is suitable for predictive parsing: studying one input token at a time is always enough to decide how to expand any nonterminal. For example, if a predictive parser is looking to expand C, and sees token **seq**, then it should apply the production $C \longrightarrow \textbf{seq}\ B$; if it sees **par**, then production $C \longrightarrow \textbf{par}\ B$ is right.

All the information about what steps a predictive parser should take can be summarised in a *parse table* such as the following.

G_1	**seq**	**par**	value	**end**
C	$C \longrightarrow \textbf{seq}\ B$	$C \longrightarrow \textbf{par}\ B$	$C \longrightarrow \mathsf{value}$	
B	$B \longrightarrow C\ B$	$B \longrightarrow C\ B$	$B \longrightarrow C\ B$	$B \longrightarrow \textbf{end}$

This is a matrix indexed by nonterminals and terminals, with each entry being a production from the grammar. For example, looking up (B,**end**) we get $B \longrightarrow \textbf{end}$, which means that is the production to apply to expand B when the next input token is **end**. More interestingly, it shows that a parser should expand nonterminal B using $B \longrightarrow C\ B$ on seeing either **seq**, **par**, or value; the point being that the C can then be further expanded using productions from the top row of the table.

Empty cells in the table show that some nonterminal/terminal combinations should never arise: here a circuit C can never begin with the keyword **end**.

Notice that every production in the C row has C on its left-hand side; similarly every production in row B expands B. This is always true of predictive parse tables, so to save space in later ones we shall write only the right-hand side of productions. Similarly, we leave out any columns where every cell is empty.

Here is another grammar G_2 and corresponding parse table. This describes a language of nested lists like {x, (x,x), x} and (((x))). Every list must have at least one element, and can use both parentheses () and braces {} provided they always match.

$$\begin{array}{llll} \mathit{term} \longrightarrow \mathit{braces} \mid \mathit{parens} & \mathit{braces} \longrightarrow \{\mathit{list}\} & \mathit{list} \longrightarrow \mathit{term}\ \mathit{rest} & \mathit{rest} \longrightarrow , \mathit{list} \mid \varepsilon \\ & \mathit{parens} \longrightarrow (\mathit{list}) & \mathit{list} \longrightarrow \mathrm{x}\ \mathit{rest} & \end{array}$$

G_2	{	}	(	)	x	,
term	*braces*		*parens*			
braces	{*list*}					
parens			(*list*)			
list	*term rest*		*term rest*		x *rest*	
rest		ε		ε		, *list*

When a grammar is suitable for predictive parsing, a parse table like this contains all the information necessary to carry out parsing. In Note 12 we shall see how to compute parse tables; but first we look at how they can be used to drive an automatic parser.

Table-driven parsing

We consider a predictive parsing engine that uses a stack to keep track of its activity. The stack holds a sequence of terminal and nonterminal symbols, which record what work remains to be done. (More precisely, the stack records what we are expecting the remainder of the input to look like.) As parsing proceeds, the engine adds and removes symbols at the top of the stack. The engine also has access to the input string which it consumes one token at a time.

As an example, suppose we wish to parse the sentence (x) using the grammar G_2. We can illustrate the initial state of the parser as follows.

Input	Stack
(x)	*term*

At the start of parsing the working stack holds only the start symbol. The parse engine now takes repeated steps according to the following instructions.

- If the symbol at the top of the stack is a terminal, then it should match the current input symbol. If so, then discard both tokens and proceed. If they do not match, report an error of the form “Expected ‘a’ but found ‘b’ ”.
- If the symbol at the top of the stack is a nonterminal A, and the current input symbol is a, then look up the entry (A,a) in the parse table. This should give an appropriate production $A \longrightarrow t$: replace A by t on top of the stack and proceed.

  If the entry (A,a) is empty, signal an error of the form “When trying to process an ‘A’, found ‘a’ and no production applies”.

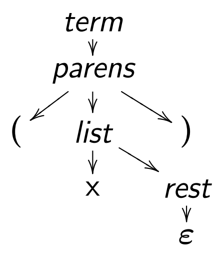
- If the end of the input is reached before the stack is empty, signal an error of the form "End of input reached where *s* expected", where *s* is the symbol at the top of the stack. Likewise, if the stack is emptied before the end of input is reached, signal an appropriate error message.

With a valid sentence, this will continue until the stack is emptied and the whole input string is consumed (with these two events happening at the same time). If the sentence is invalid, then one of the errors above will be reported.

Applying this algorithm to the example above, the parse engine takes the following steps. The top of the working stack is to the left, and as we go along we show the productions used and the derivation this produces.

Input	Stack	Productions	Derivation
(x)	*term*	*term* ⟶ *parens*	*term* ⇒ *parens*
(x)	*parens*	*parens* ⟶ (*list*)	⇒ (*list*)
(x)	(*list*)		
x)	*list*)	*list* ⟶ x *rest*	⇒ (x *rest*)
x)	x *rest*)		
)	*rest*)	*rest* ⟶ ε	⇒ (x)
)	)		

Parse tree



You should work through each line of this table, checking against the instructions above to see what happens at every step. Notice how each production corresponds to an unfolding of the parse tree; at the end we can read the original sentence off round its fringe.

At every point during parsing, the working stack holds a sentential form that should match the remaining input. This is the extensible memory that makes our parser more powerful than any finite state machine. For instance, the stack here records how many brackets of each kind still need to be matched. A machine that can use a stack in this way is sometimes known as a *pushdown automaton*.

Semantic actions

When a parser is part of some complete language processor, we usually want a more detailed result than just success or failure. A general approach to this is to attach a *semantic action* to each production. When the parsing engine applies a production, it carries out the relevant semantic action, and together these compute the desired output value. Often, semantic actions are used to build an *abstract syntax tree*, a stripped-down version of the parse tree. This discards all the syntactic fluff of semicolons, parentheses and extra nonterminals, and contains only the bare structure necessary for further interpretation.

John Longley 2002, Ian Stark 2001