NOTES ON PARSING (for *Computerphile*)

David F. Brailsford
Ronald D. Knott †

1. Introduction

These notes are intended as a brief introduction to the topic of parsing Chomsky Type 2 languages. What is set out here is by no means a comprehensive survey. Further details can be found, for example, in Cohen [1] and in the 'Dragon Book' (so called because of the picture of a red dragon on the front) about compilers, by Aho, Hopcroft and Ullman [2].

The major aim is to give some idea of how the deterministic subset of Type 2 languages (those that can, equally, be modelled as a Deterministic Pushdown Automaton or DPDA) can be further subdivided into a nested hierarchy of grammars for those languages (these grammar types, as we shall see, have names such as LL(1), LR(1), LALR(1) and so on). These grammar types, and the *parsers* that cope with them, form an important part of the study of languages and compilers.

Recall at the outset that Type 3 (regular) grammars need no 'memory' ; they have an equivalent modelling as *Regular Expressions* or as *Finite State Automata* and even if the automaton one draws is for a non-deterministic machine (NFA) it is always possible to turn this into a deterministic equivalent (DFA). Thus, the initial task in a 'compiler' for some given language is to gather the input characters into meaningful groupings (called *tokens*) e.g. for reserved words such as **`struct`**, or identifiers such as `i`, `count1` and so on. This process is called *lexical analysis* or *tokenizing* and the implementing software is called a *lexical analyzer*, or a *tokenizer*.

It is the tokenizer's job to make its recogniser be a DFA rather than a NFA. This may involve adding more states to the NFA (i.e. adding further rules to the regular expressions that recognise the tokens). Chomsky Type 2 grammars have an equivalent automaton formulation involving a *stack* (often called a *pushdown store* in older texts). Hence the equivalent automaton formulation is called a *pushdown automaton* (PDA). We shall see that it is *often* possible to make this automaton be deterministic but, unlike Type 3, success cannot be guaranteed.

The availability of a stack for Type 2 parsing makes grammars of this variety natural contenders for being able to parse the *nested block structures* that so often occur in programming languages. Equally, a Type 2 grammar can parse *palindromes* (strings of characters that read the same backwards as forwards) but, as we shall see, generalised palindromes of arbitrary length, although of Type 2, will still defeat any attempt to create a **deterministic** parser. (Note that Type 3 cannot cope with generalised palindromes at all; a specific DFA could certainly be created for words such as 'rotor', 'refer', 'deed' etc. but every single one would have to be in the DFA as its own special case. The DFA would be enormous and inelegant and there would be no ability to cope with new, or arbitrary-length, palindromes.)

---

† Note from DFB: Ron Knott was one of my grad. students in the late 1970s. His PhD work was centred around the language Algol68 and involved using top-down (J. M. Foster's SID) and bottom-up (Steve Johnson's YACC) parser-generators, to create compilers for Algol68. The first version of these notes was drawn up by Ron and they have been tweaked and extended by me in several places since that time, to such an extent that — while thanking him again for his excellent efforts — the responsibility for any errors and omissions lies firmly with me.

2. A simple Type 2 parsing example

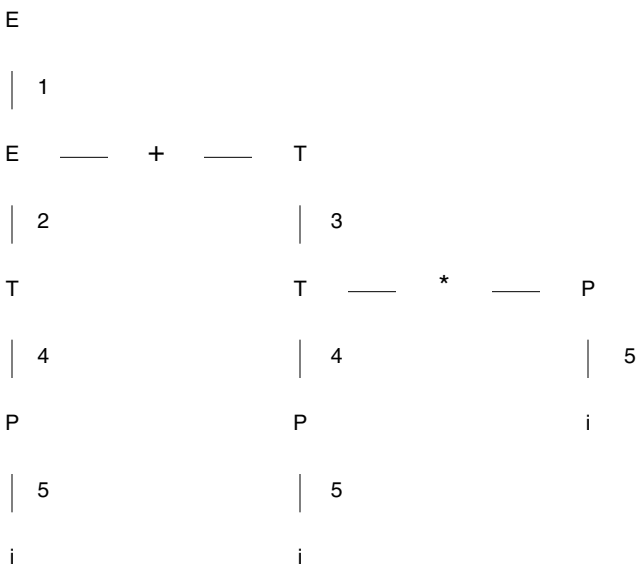
In the example that follows we shall use a modified Backus-Naur Form (BNF) for the syntax of the grammar, similar notation in fact to that required by the YACC parser-generator.

Here is a grammar for simple arithmetic expressions with `i` being used to denote numbers or identifiers at the leaves of the tree:

```
E: E '+' T              1
 | T ;                  2
T: T '*' P              3
 | P ;                  4
P: 'i'                  5
 | '(' E ')'            6
```

The production rules are numbered on the right for ease of reference. The vertical bar 'l' denotes 'or' and it separates alternative right-hand sides for the various rules in the grammar. The notation also uses primes (single quotes) to enclose terminal characters. The above grammar is unambiguous, i.e. each string in the language it defines has only **one** parse tree.

Parsing `i+i*i` gives :-



There are many ways to arrive at this tree for the given string, but the tree itself is unique. To write down the production numbers used in producing the tree is to produce the **derivation** of the string. Again, there are many ways to write down a derivation (which constitutes an informal 'proof' that the string belongs to the language defined by the grammar.

2.1. Top-down — expand the leftmost non-terminal first

```
E
E+T     1
T+T     2
P+T     4
i+T     5
```

```
i+T*P   3
i+P*P   4
i+i*P   5
i+i*i   5
```

The derivation `12453455` is called the **leftmost derivation**. Notice carefully that this leftmost derivation corresponds to **postorder traversal** as already covered in a previous video.

2.2. Top-down — expand rightmost non-terminal first

```
E
E+T     1
E+T*P   3
E+T*i   5
E+P*i   4
E+i*i   5
T+i*i   2
P+i*i   4
i+i*i   5
```

Here `13545245` is the **rightmost derivation**.

2.3. Bottom up — reducing leftmost set of symbols first

```
i+i*i   5
-

P+i*i   4
-

T+i*i   2
-

E+i*i   5
  -

E+P*i   4
  -

E+T*i   5
    -

E+T*P   3
  ---

E+T     1
---

E
```

The symbols reduced on each occasion are underlined. They are called the **handles** of the parse. Note that the derivation produced this time is `54254531` which is the *reverse* of the rightmost derivation and is called the **right parse**.

The 'natural' parse for a top-down parsing method is the leftmost derivation, called the **left parse**, whereas for a bottom-up method it is the *reverse* of the rightmost derivation which is the

most natural, and this is called the **right parse**. (And these derivations correspond to **preorder traversal** already covered in a previous video),

3. Producing derivations directly from the parse tree

The derivations may be obtained directly from the parse tree as follows. From the root of the tree, trace round the outline of the tree starting on the left, writing down the production numbers passed on vertical branches on the way down. This gives the leftmost derivation. The rightmost derivation is found by starting on the right of the root and then using the same method. This rightmost derivation is the *reverse* of the right parse.

4. EXERCISES

1. For the grammar above, give the parse tree and the left and right derivations of:
   (a) `i+i*i`
   (b) `i*i*i`
2. For the grammar above which strings correspond to
   (a) The left parse `11245345545`
   (b) The right parse `54254531541`
3. Give a set of syntax rules for a real number, for which all the following are legal examples

   +3.000 .05 54. -34567

   A number may be optionally followed by an exponent of the form: `e` followed by an optionally signed integer.
4. Give a syntax for a Roman Numeral. (Recall: `I=1; V=5; X=10; L=50; D=500 C=100; M=1000`). Check that your syntax allows the following:

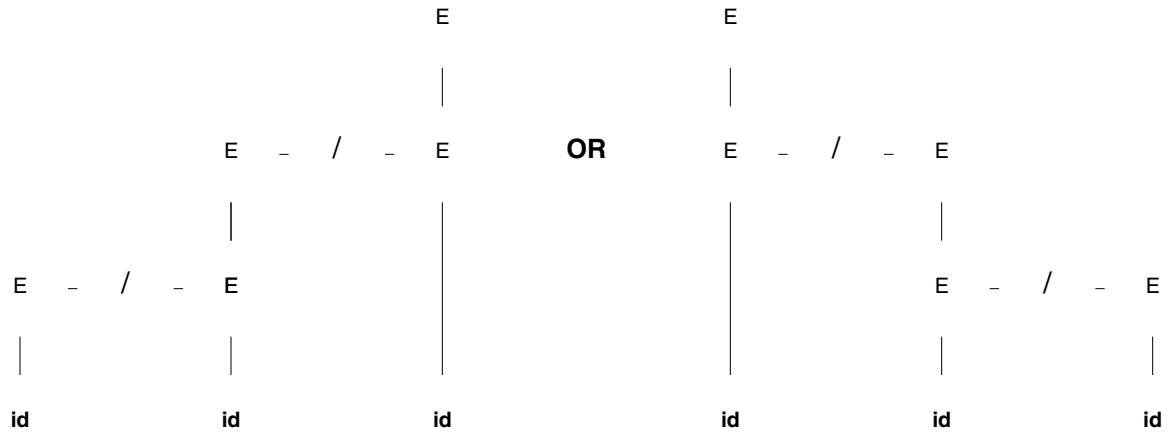
   `MCMLXXVIII, MIX, MMMMMMMMMXCIX`

5. Ambiguous Grammars

A grammar is said to be *ambiguous* if it can derive more than one parse tree for a given input string. Consider the following grammar for some very simple arithmetic expressions:

`E : E + E | E*E | E / E | (E) | -E |` **`id`**

where **`id`** is a terminal symbol standing for some arbitrary identifier. This grammar is ambiguous because, for instance, the string **`id / id / id`** can be parsed as:



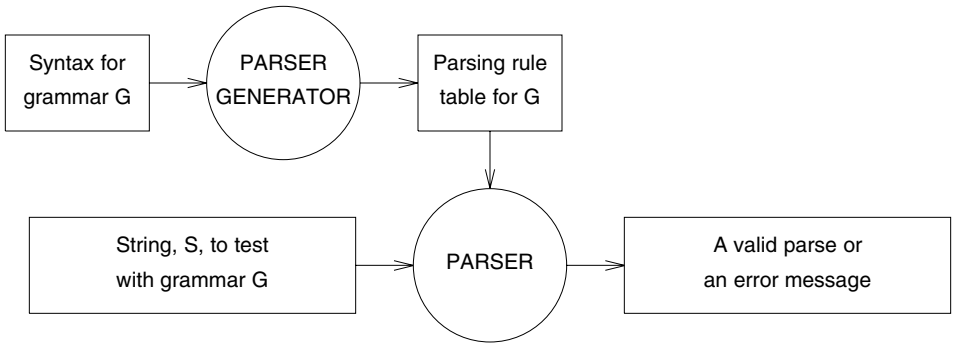
Sometimes an ambiguous parse does not alter the meaning of the string, or 'program', that is being analysed. But in the above case, if we assume `/` denotes integer division and that our 3 successive **`id`**'s have values of 8, 4 and 2 then the left-hand diagram represents `(8/4)/2` with a value of `1` whereas the right-hand one represents `8/(4/2)` with a value of `4`.

Notice that the unambiguous grammar of section 2 gets around the problem of ambiguity by embedding operators of high precedence (such as multiply) further down in the grammar, and hence low down in the parse tree, whereas the low priority operators such as + appear in earlier rules and hence at the top of the tree near the root. Fully developed grammars of this sort are often called *operator precedence* grammars.

Most parser-generators, YACC included, if presented with ambiguous grammars of the type discussed in the present section, have certain *disambiguating rules* to circumvent problems. YACC's convention, for successive operators of equal precedence at the same 'level' in a grammar is to assume that associativity goes left-to-right i.e. for the numeric example discussed `(8/4)/2` is the preferred parse.

6. General Parsing Algorithms

Algorithms exist which will produce a parser for legal strings in a given grammar provided that the grammar satisfies certain conditions. We shall not look in any great detail at the algorithms involved but concentrate instead on the organisation of the parsers that two such systems produce. Such **parser generators** form the heart of many present-day compiling systems.



7. General top-down parsing

Let us take the most general Type 2 case i.e. where the grammar we are given is certainly of Type 2 but may not be deterministic. For such a given grammar, we start with the sentence type for which we want to test an input string. This forms our first *goal* phrase-type. We then successively expand this goal, taking each production of it in turn to see if we can match the input string with it. The current goal is replaced by a new set of goals which are the successive elements of the production for the original goal. If a goal is a terminal symbol it must match the next symbol of the input which is then discarded and the next symbol from the input is read. We report 'success' for that particular goal.

If the goal is another phrase type (i.e. a non-terminal symbol) then we apply the above process to it, recursively. If any production delivers a 'fail' when we try to match to the next available symbols then we must **backtrack** to the non-terminal for which we substituted this production and try another alternative, restoring any input already read. If we exhaust all productions and still have not found a 'success' we report a 'fail' for the non-terminal whose productions we have been trying and backtrack one more level.

We must have 'success' reported for all elements of a production before we can report 'success' for the entire production – and therefore for the non-terminal we have expanded. If we end

up with the goal stack empty (we pop off goals as they are achieved) and at the same time all input has been read we can report 'accept' for the string — it is a legal string in the language defined by the grammar. Otherwise we report a 'syntax error'.

The left parse is found by emitting the production numbers whenever we replace a goal by one of its productions (we emit this number **before** we test the production). With backtrack it may be necessary to try each production in turn as presented in the grammar. Let us try to parse the string `aabbb` within the grammar:

```
S: 'a' T T      1
  |'b' ;        2
T: 'b' S S      3
  | 'a' ;       4
```

The parser actions are as follows:

Action	Goal Stack	Input Unread	Output
Initialize	S	aabbb	
try 1	aTT	aabbb	1
success	TT	abbb	1
try 3	bSST	abbb	13
fail, backtrack	TT	abbb	1
try 4	aT	abbb	14
success	T	bbb	14
try 3	bSS	bbb	143
sucess	SS	bb	143
try 1	aTTS	bb	1431
fail, backtrack	SS	bb	143
try 2	bS	bb	1432
success	S	b	1432
try 1	aTT	b	14321
fail, backtrack	S	b	1432
try 2	b	b	14322
success			14322
ACCEPT			

Thus the left parse for `aabbb` in this grammar is `14322`.

8. Deterministic top-down parsing

It is not hard to see that, for the particular grammar above, we could have produced a much more efficient parser. Notice that in the previous grammar we can always determine the correct production to choose to expand any terminal goal by simply examining the next (single) input character. If the goal is `S` and the next input symbol is `a` then choose production `1`. If it is `b` choose `2`. Any other symbol produces a failure and the string is illegal. Such a parsing method is **deterministic** since it never involves backtrack, no matter what (legal) string of the language it is parsing. But notice that in order to avoid backtrack we have had to indulge in **lookahead** — by one symbol in this particular case. The parser of this deterministic type, for the grammar above works like this:

```
initialize:     Put S on the goal stack
Then:           If current goal is S

                        if next input=a, emit 1, replace S by TT, Read.
Repeat
                        if next input=b, emit 2, pop S.  Read. Repeat

                If current goal is T
                        if next input=a, emit 3, replace TS by TT, Read. Repeat.
                        if next input=b, emit 4, pop T. Read. Repeat.

        If stack is empty and all input read report 'ACCEPT'
        else report 'SYNTAX ERROR' and attempt to recover or, simply, stop.
```

For some grammars we cannot produce such a deterministic parser. The grammar above is one of a special class called LL(1) meaning that it requires a **l**eft-to-right scan of the input text, produces a **l**eft parse and needs **1** symbol of lookahead to make the correct choice. Software that attempts to transform grammars to LL(1) form and to build a parser was first developed by Lewis and Stearns in 1968 and, independently, by J. M. Foster shortly after that (see Dragon Book).

Some grammars of the LL variety need more than one symbol of lookahead in order to decide which production rule to use next. These are called the *LL*(*k*) grammars where *k* denotes the number of symbols of lookahead that are needed. Consider:

```
S: 'a' 'b' T
  | 'a' 'c' T ;
T: 'd' ;
```

This is LL(2) since we need to look beyond the next `a` symbol when we start to expand `S` to see if a `b` or a `c` follows. Remember that if we are to output the left parse we need to output the production rule number **before** we test that rule.

So, if we see `ab` next, we expand by production 1; if we see `ac` we expand by production 2. Since two symbols of lookahead suffice the grammar is LL(2). Note that the grammar:

```
S: 'a' U 'd' ;
U: 'b' | 'c' ;
```

describes the same language as the grammar above but this new grammar **is** LL(1). This shows that the LL(k) property is a property of a **grammar** and not the **language**. This means that we might be able to describe a language by an alternative grammar, which is LL(1), if the original is not acceptable to our LL(1) parser-generator.

Just in case you think that all Type 2 grammars could be made LL(k), for some suitably large *k*, consider:

```
S: R | T ;
R: 'a' R | 'b';
T: 'a' T | 'c';
```

The language described here is $a^n\ b$, $a^n\ c$ $n \geq 0$, definitely of Type 2, but the grammar above is not LL(k) for any finite *k*. The reason is that we need to look beyond the `a`'s (and there may be an infinite number of these) to see if `b` or `c` occurs before we can tell whether to use production 3 or production 5. Interestingly an equivalent LL(1) grammar can be found for this same languaget Can you work out what it is?

8.1. Left recursion and LL parsers

Consider the grammar:

```
S: T U ;
T : T 'b' | 'c' ;
U: .....
```

The top-down method gets into an infinite loop when it tries to expand the goal `T` since its first expansion is `T` again (i.e the rule for `T` is left recursive). Thus the top-down parsing method *fails for left-recursive grammars.* Moreover, it is defeated not just by the directly left-recursive rule shown above but also by hidden left recursion as in:

```
S : T U | R ;
T : S 'a' ;
U: ...
```

If we expand by rule 1 it sends us to rule 3 and when we expand `T` we meet `S` again To produce a left parse we must emit the production number used to expand `S` *before* we test whether it works or not.

8.2. Removing left recursion

Parser-generators of the LL variety have to attempt a clean-up of the input grammar in two respects. Firstly, productions which have ε (i.e. the null string) as an option on the RHS of a rule can cause great difficulties when automating a top-down parser. Fortunately an algorithm due to Bar-Hillel, Perlis and Shamir enables these 'nullable' productions to be written in a more amenable form. The next stage seeks to eliminate any 'direct' or 'hidden' left recursion from the rules. This is much trickier but an algorithm due to Sheila Greibach can be tried, so as to bring the grammar into *Greibach Normal Form* with left recursion removed but still able to generate precisely the same language as the original grammar.

After this has been done an attempt is made to transform the grammar into LL(1) form. Unfortunately, it is in general **undecidable** whether an arbitrary CFG can be made deterministic in this way and occasionally the parser-generator can loop and crash in making the attempt.

It turns out that another class of parsers, called LR, which work 'bottom up' rather than top-down are slightly more powerful than LL parsers. More importantly, right recursive production rules are not as deadly for bottom-up parsers as left recursive ones are for top-down parsers. We examine this in more detail in the next section.

9. GENERAL BOTTOM-UP PARSING

We now turn our attention to another parsing method. In bottom-up parsing the aim is, starting with the input string, to **reduce** it, using the productions of the grammar, until we end up with the single phrase-type which we are testing for, i.e. the sentence-type.

The general method is to find the leftmost set of symbols of input which form the complete right-hand-side (rhs) of a production, and then replace this 'handle' by the left-hand-side (lhs) of that production. The main problem is identifying the handles at any point. If we can always do this without having to retry (backtrack), then the parse is deterministic. The LR(k) grammars are the most general ones that are amenable to the deterministic bottom-up approach, and YACC is an example of an LR(1) Parser Generator (more precisely it is LALR(1) — see later).

We now look in a little detail at the organisation of a typical parser produced by such a system, but not at how these parsers are produced from the grammar.

LR(k) means

- **L**eft to right scan on input
- Producing a **R**ight Parse
- Correct decision on handle detection being determined by at most **k** input symbols.

9.1. The LR(k) Parsing Method

We need a stack on which to store input symbols ready for reduction. Since we are to produce the right parse, we must reduce when a suitable handle appears on the top of the stack.

An LR(1) Parser

We look in detail at the parsing method the YACC LALR(1) parser generator uses in the parsers it produces. The grammar rules have a **state** associated with each position the parser may find in the rules. The states correspond to places between terminals and/or non-terminals in the RHS's of productions.

We have a stack on which we perform the reductions, and we tag each element on it with the state that gave rise to it. The current state is the top state on the stack. We successively look up a state in a table and obey one of 4 types of action according to the entry. Then we move into a new state and repeat. The 4 types of action are:

(a) *Shift* the input symbol on to the stack, together with the new state number (found from the table).

(b) A handle is on the stack. *Reduce* it according to the production number given in the table. Find the new state from the table and stack it too.

(c) Accept. The string has been successfully parsed.

(d) Fail. The string is illegal. Halt – or try to recover.

Since the method is LR(1), we need only examine the current state and the next single input character to decide on the action to take. The parser generator will test for this condition when it produces the parser for a given grammar.

To illustrate, we give the parser for the grammar:

```
L  :  L  ','  E          1
   |  E  ;               2
E  :  'a'                3
   |  'b' ;              4
```

The language is a list of `a`'s or `b`'s separated by commas. First we augment the grammar with one extra rule, at the 'top' of the syntax:

```
Accept  :  L  '$end' ;  0
```

where `$end` marks the end of the input string. The states are as follows:

```
Accept  :       L       $end;
L       :       L       ','     E
         |      E       ;
E       :       a
         |      b       ;
```

The parsing-action table produced by YACC is equivalent to:

	Terminals				Non-terminals	
State	a	b	,	$end	L	E
0	s3	s4	#	#	1	2
1	#	#	s5	ACC	#	#
2	#	#	r2	r2	#	#
3	#	#	r3	r3	#	#
4	#	#	r4	r4	#	#
5	s3	s4	#	#	#	6
6	#	#	r1	r1	#	#

`s`*n* means Shift input symbol onto the stack. New state is denoted by *n*.
`r`*n* means Reduce according to production number *n*.
`ACC` means Accept input as valid.
`#` means 'syntax error' if this entry is uncovered in this parse.

Method:

1. Initialise:

   Stack `$start` (an imaginary terminal marking the start of the input string, like `$end`) together with state number U.

2. Parsing actions:

   Act on the current entry from the table found from

   (i) the current state.

   (ii) the current input symbol to be read (terminal) (The non-terminal entries are only used after a reduction).

   Action:

   (a) `S` *n*:
   Shift the current input symbol onto the stack, and move into state denoted by *n*. Get the next input symbol.

   (b) `R` *n*:
   Emit the production number *n*. This will give the right parse if we succeed in parsing the input. Reduce the stack according to the production. i.e. the RHS of the production will be found on the stack. Replace it by the lhs of the production. The new state is found from the table entry under the new non-terminal we have just stacked, for the last state number visible on the stack.

   (c) `#`:
   Syntax error. Halt.

   (d) `ACC`:
   Parse is successful. The numbers emitted form the right parse.

To parse `a , b , a` let < denote the beginning of that input and > the end (`$begin` and `$end`). The stack is represented in 2 layers, the top being terminal and non-terminal symbols, the second row being the state associated with each symbol on the top row. The current state is the state number at the right hand end of row 2.

Action	Stack	Input unread	Output
Initialize	<	a,b,a>	
	0		

```
s3     <a          ,b,a>
       0 3

r3     <E          ,b,a>   3
       0 2

r2     <L          ,b,a>   32
       0 1

s5     <L ,         b,a>   32
       0 1 5

s4     <L,b           ,a>  32
       0 1 5 4

r4     < L , E        ,a>  324
       0 1 5 6

r1     < L             a>  3241
       0 1
s5     < L ,           a>  3241
       0 1 5

s3     < L , E          >  32413
       0 1 5 3

r3     < L , E          >  32413
       0 1 5 6

r1     < L              >  324131
       0 1

Accept
```

Thus the right parse for `a,b,a` is `324131`.

9.2. Other LR(k) grammars

From the above, the LR(k) method works even for left recursive grammars. Recursive grammars in any form are not a problem to LR(k) parser-generators, although right-recursive grammars can lead to very large stack sizes in the parser: essentially input symbols have to be stacked until a handle appears.

Thus, we can now revisit the troublesome non-LL(k) grammar (for convenience, lower case letters represent terminal symbols; upper case, non-terminals):

```
S  :  R  :  T;
T  :  a  T  |  c;
R  :  a  R  |  b;
```

and we find that it is now acceptable. It is LR(1) since we can arrange to stack symbols (`a`'s) until we see `b` or `c`. This is consistent with a right parse, and we reduce by production 4 or 6 first, before 3 or 5. (Hint: consider the parse tree for `aa...ab` or `aa....ac`)

For the following grammar:

```
S  :  R  b  c  |  T  b  d;
R  :  a;
T  :  a;
```

whenever we read `a` and have shifted it onto the stack, we cannot decide whether to reduce the `a` to an `R` by rule 3 or reduce by rule 4 since both allow `b` as the next symbol. However, using 2 symbols of input resolves the dilemma. This grammar is LR(2). Note that

```
S  :  a  b  U;
U  :  c  |  d;
```

is LR(1) yet defines the same language. LR(k)-ness is, again, a property of the **grammar** and not of the **language** — just as for LL(k).

The following is not LR(k) for any k.

```
S  :  a  T  c;
T  :  b  T  b  |  b  ;
```

since we cannot decide when to reduce the middle `b` by production 3 until we have seen the termination `c`, which may be infinitely far ahead. Notice that this grammar is very similar indeed to PALINDROME — the classic non-deterministic Type 2 language. Appendix A gives a fuller analysis of what happens when you input this grammar to YACC. Essentially YACC cannot always recognise that it is faced with a Type 2 grammar it cannot handle! It applies certain rules about shift/reduce conflicts and is convinced that it has produced a parser of some sort for 'palindrome'. Only when you feed in a legal string and the parser declares it to be illegal do you realise that this is one of the (very) few Type 2 grammars for which YACC cannot compile a valid parser.

9.3. The LR hierarchy

One factor which arises in LR parsing is how parser conflicts are handled. Situations can arise where a potential handle appears on the parser stack (and is apparently ready to be reduced) but it may be that if another symbol were to be shifted from the input stream progress could be made towards building up a longer handle. This is called a *shift-reduce* conflict. The other situation is where two or more reductions are possible for a given handle on the stack; this is a *reduce-reduce* conflict. The best way to minimise such conflicts is to construct a canonical LR parser along the principles first laid down by Knuth in 1965 (see Dragon Book). The problem here is that parsers of this sort (for a language of the complexity of C, say) would have internal parser tables with many thousands of entries Certain simple LR grammars (SLR) produce parsers with many fewer states, and with no parser action conflicts, but they are sometimes not powerful enough to parse constructs that naturally occur in programming languages.

LALR (**L**ook **A**head LR) parsers have the same number of states as an SLR parser, for the same grammar, but by allowing lookahead the power of the parse method is greatly increased, for constructs naturally appearing in programming languages. An alternative way of looking at the problem is that SLR and LALR use methods that *combine states* from a full canonical LR parser in order to save table space and to speed up parser action. It can be shown that this merging of states will never introduce new *shift-reduce* conflicts but it can produce new *reduce-reduce* conflicts.

YACC is a classic example of a parser-generator that produces LALR parsers (if possible) from the given set of grammar rules. YACC's convention is that faced with a shift-reduce conflict it will always shift; if faced with a reduce-reduce conflict it will reduce by the earliest possible rule in the grammar.

10. Comparison of LL and LR Parsing Methods

Note that the general LR requirement is that we be able to recognize the occurrence of a RHS of a production rule having seen what is derived from that production and having stored state information about it on the stack. This is a much less stringent requirement than in top-down parsing where we need to detect the apparent use of a production seeing only the first sysmbol that it derives. Intuitively therefore it is not surprising that the class of grammars accessible to LR techniques is wider than that accessible to LL techniques [*]. Let us summarise as follows:

(1) Both LL and LR need a stack which may grow indefinitely according to the input read.

(2) Both are deterministic, so the parsers are relatively fast and efficient.

(3) Both detect a syntax error at the earliest point in the input string.

(4) Neither is sufficiently general to be used for an arbitrary Type 2 grammar that can be writen in BNF.

(5) However, every LL(k) grammar has an equivalent LR(k) grammar of the same *k*, but there are LR(k) grammars for which there are no LL(k) equivalents i.e. LR(k) methods apply to a wider class of grammars than LL(k).

(6) Every LR(k) grammar has an equivalent LR(k-1) grammar ($k>1$) so an LR(1) parser-generator is sufficient for all LR grammars. Similarly a LL(1) parser suffices for LL(k).

(7) Most programming languages can be described with an LALR(1) grammar (see above).

(8) Left recursion cannot be dealt with be an LL(k) parser, and such grammars have to be re-written before a parser of the LL(k) type can be produced. The Greibach transformations outlined above make removal of left recursive rules difficult but possible. But LL parser generators can still fail, even after removal of left recursion, if the basic grammar is not capable of being transformed into the LL subset.

(9) LR parsers can cope with both left and right recursion but will still fail for the 'self-embedding' kind of recursion found in PALINDROME.

References

1. Daniel I . A. Cohen, *Introduction to Computer Theory (revised edn.),* John Wiley Inc., 1991. ISBN 0-471-54841-3

2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles Techniques and Tools,* Addison-Wesley, 1986. ISBN 0-201-10194-7 (The 'Dragon Book')

---

* Interestingly, one could in principle choose to parse a program backwards (!) i.e. to create a right-to-left scanner. Not surprisingly, RR parsers have the same power as LL ones; RL parsers have the same power as LR.

APPENDIX A

YACC's behaviour with innately non-deterministic Type 2 grammars

In a previous section we encountered a grammar for $ab^{2n+1}c$ which we shall call TESTGRAMMAR. Let us first consider a grammar for ab^nXb^nc (TESTGRAMMARX). We have already noted that these grammars are similar to PALINDROME and PALINDROMEX, repectively, and, once again, in TESTGRAMMARX the `X` unequivocally marks the mid-point of the string.

Not surprisingly, YACC has no problem in producing a parser for this deterministic grammar. Remember that YACC requires a lexical analyser front-end procedure called `yylex()`. and the easiest way to provide this is to write one in LEX itself. In this case the LEX input is amazingly simple:

```
%%
"a"     {ECHO;return('a');}
"b"     {ECHO;return('b');}
"c"     {ECHO;return('c');}
"X"     {ECHO;return('X');}
"&"     {ECHO; return('&');}
"\n"     {ECHO; return('\n');}
%%
```

Our language involves only the letters `a`, `b`, `c`, and possibly `X` (for TESTGRAMMARX). The `\n` token allows us to (in conjunction with the YACC grammar later on) to try several possible inputs separated by newlines. The symbol `&` will be on a line of its own and will denote end of input. The quote marks are essential for the `&` because it will be interpreted as LEX metanotation otherwise (similarly `\` is metanotation so `\n` needs quote marks. But quoting things never hurts — and so I've quoted all terminal symbols in the above rules. The `ECHO` is a LEX library macro that echoes back to your terminal each character as it is read.

Now here is the YACC grammar for TESTGRAMMARX

```
%{
# include <stdio.h>
# include <ctype.h>
%}

%start list

%%
list    : /*empty*/
        |list S '\n'
        |list error '\n'
                { yyerror("Oh dear  -- syntax error. Baling out now\n");
exit(1)
;}
        ;


S       :'&'    /* & marks end of input */
                        {printf("End of input-- Bye! ");exit(0);}

S       : 'a' T 'c'
                {printf("Parsed OK!\n");}
        ;
T       : 'b' T 'b'     {printf("Rule 2. Bingo!\n");}
        | 'X'           {printf("Rule 3. Yippeee!!\n");}
        ;
%%
#include "lex.yy.c"
main() {
        return (yyparse() );

        }

yyerror(s)
        char *s;
{
        fprintf (stderr, "%s\n", s);

}
```

Things to notice are that we must inform YACC of the distinguished symbol, `list` of our grammar.The real distinguished symbol is `S` but the left recursive 'wrapper' rule for `list` enables a succession of `S`'s to be analysed one after another until a terminating `&` is read. (Question: why is the `list` rule made left recursive rather than right recursive?).

The compiled C source code for the lexical analyser is simply textually included into the C code that YACC outputs by virtue of the line `#include "lex.yy.c"`. The `main()` routine must be present to call up the parser that YACC generates. A `yyerror()` routine though not compulsory is highly advisable in order to get some idea of where the parse fails. Notice the 'actions' written in C after each of the grammar rules.
To create our complete parser we do (where `$` is the UNIX prompt):

```
$ lex testgrammar.l
$ yacc testgrammar.y
$ gcc y.tab.c -o xparser -ly -lfl
```

The C code for the parser is generated by YACC in the file `y.tab.c`. Note that the GNU C compiler (`gcc`) is used because this has full support for LEX and YACC libraries on Linux machines and these libraries are called in via the `-ll` (or `-lfl`) and `-ly` flags. The `-o parser` flag leaves the compiled binary program in the executable file `parser` rather than the annoyingly anonymous `a.out`.

So, to run the parser against some input we do:

```
$ xparser
abbbXbbbc
&
```

and the output generated is:

```
abbbXbbbc
abbbXRule 3. Yippeee!!
bRule 2. Bingo!
bRule 2. Bingo!
bRule 2. Bingo!
cParsed OK!

&
&End of input-- Bye!
```

The first line of the output is simply generated by the LEX ECHO commands. Subsequent lines are generated by the actions embedde in with the grammar rules. It's very clear that YACC has spotted the need to shift symbols until an X is encountered and the valid 'handle' is then on the stack. It then backs off recursively up the rules and correctly parses the string.

Now, edit the file `testgrammar.y` and replace the `X` in rule 3 with another `b`. Then do the following (YACC will report 1 shift/reduce conflict; notice we don't need to regenerate the lexical analyser):

```
$ yacc testgrammar.y
$ gcc y.tab.c -o parser -ly -lfl
$ parser
abbbbbbbc
&
```

The output is now:

```
abbbbbbbc
abbbbbbbcRule 3. Yippeee!!
syntax error

Oh dear  -- syntax error. Baling out now
```

The parser thinks the input is erroneous, but it isn't. Rather, the problem is that YACC sees a potential shift/reduce conflict on the `b` of rule 3 . Its algorithm of 'if in doubt, shift' makes it think it can generate a valid parser, of some sort, by shifting `b`'s until there are no more left. So it proceeds until it hits the `c` and then declares the string invalid ! What it should have done is to *reduce* once it encountered the middle `b`. But by definition it cannot know which `b` is the middle

one. This is a classic case of a grammar beyond YACC's capabilities — mercifully these are few and far between in real-life programming languages.

Final note

As stated earlier non-determinism, non-LR-ness, and so on, are usually properties of *grammars* not of *languages*. We see this, at once, because the *language* we have been considering i.e. $ab^{2n+1}c$ is not at all problematical. Indeed it doesn't even need a Type 2 grammar at all(!) — it can be done with a Type 3, as below (in YACC notation):

```
S: abT;
T: bbT
 | c ;
```

The problem was that TESTGRAMMAR chose to generate the b^{2n+1} in a self-embedding way rather than a left- or right-recursive way. In other words the central string of `b`'s is not, innately, a generalised palindrome but we chose to generate it in a self-embedding palindromic way. Now, LR and LL parser- generators have a range of transformations for simplifying input grammars (Greibach; Bar-Hillel/Perlis/Shamir; Chomsky Normal etc.) into more tractable Type 2 forms. And yet YACC signally failed to recognise that our TESTGRAMMAR was capable of becoming deterministic, and of generating *exactly the same language*, if only it was comprehensively rewritten as a left- or right-recursive regular (i.e Type 3) grammar. But we mustn't be too hard on YACC here: the problem of whether a Type 2 grammar can be rewritten as a regular grammar, to generate the same language, is very hard — so hard in fact that it is, in general, *undecidable*.

Which only leaves the question of whether there are Type 2 *languages* so innately non-deterministic that a deterministic grammar for them just cannot exist. Well, there are such languages but they lie beyond the scope of this course…

APPENDIX B

Hierarchy of Chomsky Type 2 determinism