Chapter 11

LL versus LR parsing

The parsers that were constructed using parser combinators in chapter 3 are nondeterministic. They can recognize sentences described by ambiguous grammars by returning a *list of solutions*, rather than just one solution; each solution contains a parse tree and the part of the input that was left unprocessed. Nondeterministic parsers are of the following type:

```
type Parser a b  =  [a]  ->  [ (b, [a]) ]
```

where `a` denotes the alphabet type, and `b` denotes the parse tree type.
In chapter 10 we turned to *deterministic* parsers. Here, parsers have only one result, consisting of a parse tree of type `b` and the remaining part of the input string:

```
type Parser a b  =  [a]  ->  (b, [a])
```

Ambiguous grammars are not allowed anymore. Also, in the case of parsing input containing syntax errors, we cannot return an empty list of successes anymore; intead there should be some mechanism of raising an error.
There are various algorithms for deterministic parsing. They impose some additional constraints on the form of the grammar: not every context free grammar is allowable by these algorithms. The parsing algorithms can be modelled by making use of a *stack machine*. There are two fundamentally different deterministic parsing algorithms:

- LL parsing, also known as *top-down parsing*
- LR parsing, also known as *bottom-up parsing*

The first 'L' in these acronyms stands for 'Left-to-right', that is, input in processed in the order it is read. So these algorithms are both suitable for reading an input stream, e.g. from a file. The second 'L' in 'LL-parsing' stands for 'Leftmost derivation', as the parsing mimics doing a leftmost derivation of a sentence (see section 2.4). The 'R' in 'LR-parsing' stands for 'Rightmost derivation' of the sentences. The parsing algorithms are normally refered to as $LL(k)$ or ($LR(k)$, where k is the number of unread symbols that the parser is allowed to 'look ahead'. In most practical cases, k is taken to be 1.
In chapter 10 we studied the $LL(1)$ parsing algorithm extensively, including the so-called $LL(1)$-property to which grammars must abide. In this chapter we start with an example application of the $LL(1)$ algorithm. Next, we turn to the $LR(1)$ algorithm. Finally, we show how the $LR(1)$ algorithm is used in a commonly used parser generator tool named yacc.

11.1 LL(1) parser example

The $LL(1)$ parsing algorithm was implemented in section 10.2.3. Function `ll1` defined there takes a grammar and an input string, and returns a single result consisting of a parse tree and rest string. The function was implemented by calling a generalized function named `gll1`; generalized in the sense that the function takes an additional list of symbols that need to be recognized. That generalization is then called with a singleton list containing only the root nonterminal.

An LL(1) checker
Here, we define a slightly simplified version of the algorithm: it doesn't return a parse tree, so it need not be concerned with building it; it merely *checks* whether or not the input string is a sentence. Hence the result of the function is simply a boolean value:

```
check :: String -> Bool
check input  =  run ['S'] input
```

As in chapter 10, the function is implemented by calling a generalized function `run` with a singleton containing just the root nonterminal. Now the function `run` takes, in addition to the input string, a list of symbols (which is initially called with the abovementioned singleton). That list is used as some kind of stack, as elements are prepended at its front, and removed at the front in other occasions. Therefore we refer to the whole operation as a *stack machine*, and that's why the function is named `run`: it *runs* the stack machine. Function `run` is defined as follows:

```
run  ::  Stack -> String -> Bool
run []     []     = True
run []     (x:xs) = False
run (a:as) (x:xs) | isT a && a==x  =  run as xs
                  | isT a && a!=x  =  False
                  | isN a          =  run (rs++as) (x:xs)
         where rs = select a x
```

So, when called with an empty stack and an empty string, the function succeeds. If the stack is empty, but the input is not, it fails, as there is junk input present. If both the stack and the input are nonempty, case distinction is done on `a`, the top of the stack. If it is a terminal, the input should begin with it, and the machine is called recursively for the rest of the input. If the input does not begin with the symbol expected, we've found a syntax error. In the case of a noterminal, we push `rs` on the stack, and leave the input unchanged in the recursive call. This is a simple tail recursive function, and could imperatively be implemented in a single loop that runs while the stack is non-empty.
The new symbols `rs` that are pushed on the stack is the right hand side of a production rule for nonterminal `a`. It is selected from the available alternatives by function `select`. For making the choice, the first input symbol `x` is passed to `select` as well. This is where you can see that the algorithm is $LL(\mathbf{1})$: it is allowed to look ahead 1 symbol.
This is how the alternative is selected:

```
select a x  =  (snd . hd . filter ok . prods) gram
    where ok p@(n,_)  =  n==a && x `elem` lahP gram p
```

So, from all productions of the grammar returned by `prods`, the `ok` ones are taken, of which there should be only one (this is ensured by the $LL(1)$-property); that single production is retrieved by `hd`, and of it only the right hand side is needed, hence the call to `snd`. Now a production is `ok`, if the nonterminal `n` is `a`, the one we are looking for, and moreover the first input symbol `x` is member of the *lookahead set* of this production.
Determining the lookahead sets of all productions of a grammar is the tricky part of the LL(1) parsing algorithm. It is described in section 10.2.5–8. Though the general formulation of the algorithm is quite complex, its application in a simple case is rather intuitive. So let's study an example grammar: arithmetic expressions with operators of two levels of precedence.

An LL(1) grammar
The idea for the grammar for arithmetical expressions was given in section 2.5: we need auxiliary notions of 'Term' and 'Factor' (actually, we need as much additional notions as there are levels of precedence). The most straightforward definition of the grammar is shown in the left column below.
Unfortunately, this grammar doesn't abide to the LL(1)-property. The reason for this is that the grammar contains rules with common prefixes on the right hand side (for E and T). A way out is the application of a grammar transformation known as *left factoring*, as described in section 2.5. The result is a grammar where there is only one rule for E and T, and the non-common part of the right hand sides is described by additional nonterminals, P and M. The resulting grammar is shown in the right column below. For being able to treat end-of-input as if it were a character, we also add an additional rule, which says that the input consists of an expression followed by end-of-input (designated as '#' here).

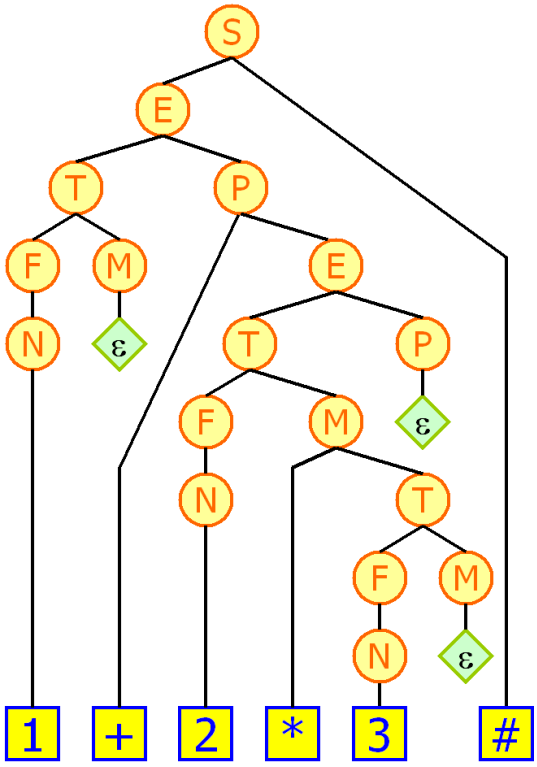
$E \rightarrow T$	$S \rightarrow E$ #
$E \rightarrow T$ + E	$E \rightarrow TP$
$T \rightarrow F$	$P \rightarrow \epsilon$
$T \rightarrow F$ * T	$P \rightarrow$ + E
$F \rightarrow N$	$T \rightarrow FM$
$F \rightarrow$ (E)	$M \rightarrow \epsilon$
$N \rightarrow$ 1	$M \rightarrow$ * T
$N \rightarrow$ 2	$F \rightarrow N$
$N \rightarrow$ 3	$F \rightarrow$ (E)
	$N \rightarrow$ 1
	$N \rightarrow$ 2
	$N \rightarrow$ 3

For determining the lookahead sets of each nonterminal, we also need to analyze whether a nonterminal can produce the empty string, and which terminals can be the first symbol of any string produced by each nonterminal. These properties are named *empty* and *first* respecively. All properties for the example grammar are summarized in the table below. Note that *empty* and *first* are properties of a nonterminal, whereas *lookahead* is a property of a single production rule.

production	*empty*	*first*	*lookahead*	
$S \rightarrow E$ #	no	(1 2 3	first(E)	(1 2 3
$E \rightarrow TP$	no	(1 2 3	first(T)	(1 2 3
$P \rightarrow \epsilon$	yes	+	follow(P)	) #
$P \rightarrow$ + E			immediate	+
$T \rightarrow FM$	no	(1 2 3	first(F)	(1 2 3
$M \rightarrow \epsilon$	yes	*	follow(M)	+) #
$M \rightarrow$ * T			immediate	*
$F \rightarrow N$	no	(1 2 3	first(N)	1 2 3
$F \rightarrow$ (E)			immediate	(
$N \rightarrow$ 1	no	1 2 3	immediate	1
$N \rightarrow$ 2			immediate	2
$N \rightarrow$ 3			immediate	3

Using the LL(1) parser

A sentence of the language described by the example grammar is `1+2*3`. Because of the high precedence of `*` relative to `+`, the parse tree should reflect that the `2` and `3` should be multiplied, rather than the `1+2` and `3`. Indeed, the parse tree does so:



Now when we do a step-by-step analysis of how the parse tree is constructed by the stack machine, we notice that the parse tree is traversed in a depth-first fashion, where the left subtrees are analysed first. Each node corresponds to the application of a production rule. The order in which the production rules are applied is a pre-order traversal of the tree. The tree is constructed top-down: first the root is visited, an then each time the first remaining nonterminal is expanded. From the table in figure 11.1 it is clear that the contents of the stack describes what is still to be expected on the input. Initially, of course, this is the root nonterminal S. Whenever a terminal is on top of the stack, the corresponing symbol is read from the input.

11.2 LR parsing

Another algorithm for doing deterministic parsing using a stack machine, is $LR(1)$-parsing. Actually, what is described in this section is known as *Simple* LR parsing or $SLR(1)$-parsing. It is still rather complicated, though.

A nice property of LR-parsing is that it is in many senses exactly the opposite, or *dual*, of LL-parsing. Some of these senses are:

- LL does a leftmost derivation, LR does a rightmost derivation
- LL *starts* with only the root nonterminal on the stack, LR *ends* with only the root nonterminal on the stack
- LL *ends* when the stack is empty, LR *starts* with an empty stack
- LL uses the stack for designating what is still to be *expected*, LR uses the stack for designating what is already *seen*
- LL builds the parse tree *top down*, LR builds the parse tree *bottom up*
- LL continuously pops a nonterminal off the stack, and pushes a corresponding right hand side; LR tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal
- LL thus *expands* nonterminals, while LR *reduces* them
- LL reads terminal when it pops one *off* the stack, LR reads terminals while it pushes them *on* the stack
- LL uses grammar rules in an order which corresponds to *pre-order* traversal of the parse tree, LR does a *post-order* traversal.

LL derivation

rule	read	↓stack	remaining
		S	1+2*3
$S \to E$		E	1+2*3
$E \to TP$		TP	1+2*3
$T \to FM$		FMP	1+2*3
$F \to N$		NMP	1+2*3
$N \to$ 1		1MP	1+2*3
read	1	MP	+2*3
$M \to \epsilon$	1	P	+2*3
$P \to$ +E	1	+E	+2*3
read	1+	E	2*3
$E \to TP$	1+	TP	2*3
$T \to FM$	1+	FMP	2*3
$F \to N$	1+	NMP	2*3
$N \to$ 2	1+	2MP	2*3
read	1+2	MP	*3
$M \to$ *T	1+2	*TP	*3
read	1+2*	TP	3
$T \to FM$	1+2*	FMP	3
$F \to N$	1+2*	NMP	3
$N \to$ 3	1+2*	3MP	3
read	1+2*3	MP	
$M \to \epsilon$	1+2*3	P	
$P \to \epsilon$	1+2*3		

LR derivation

rule	read	stack↓	remaining
			1+2*3
shift	1	1	+2*3
$N \to$ 1	1	N	+2*3
$F \to N$	1	F	+2*3
$M \to \epsilon$	1	FM	+2*3
$T \to FM$	1	T	+2*3
shift	1+	T+	2*3
shift	1+2	T+2	*3
$N \to$ 2	1+2	T+N	*3
$F \to N$	1+2	T+F	*3
shift	1+2*	T+F*	3
shift	1+2*3	T+F*3	
$N \to$ 3	1+2*3	T+F*N	
$F \to N$	1+2*3	T+F*F	
$M \to \epsilon$	1+2*3	T+F*FM	
$T \to FM$	1+2*3	T+F*T	
$M \to$ *T	1+2*3	T+FM	
$T \to FM$	1+2*3	T+T	
$P \to \epsilon$	1+2*3	T+TP	
$E \to TP$	1+2*3	T+E	
$P \to$ +E	1+2*3	TP	
$E \to TP$	1+2*3	E	
$S \to E$	1+2*3	S	

Figure 11.1: LL and LR derivations of 1+2*3

An LR checker

for a start, the main function for LR parsing calls a generalized stack function with an empty stack:

```
check' :: String -> Bool
check' input  =  run' [] input
```

The stack machine terminates when it finds the stack containing just the root nonterminal. Otherwise it either pushes the first input symbol on the stack ('Shift'), or it drops some symbols off the stack (which should be the right hand side of a rule) and pushes the corresponding nonterminal ('Reduce').

```
run'  :: Stack -> String -> Bool
run' ['S'] []     = True
run' ['S'] (x:xs) = False
run' (a:as)(x:xs) = case action of
                      Shift      -> run' (x:       as)  xs
                      Reduce a n -> run' (a:drop n as) xs
                      Error      -> False
              where action = select' as x
```

In the case of LL-parsing the hard part was selecting the right rule to expand; here we have the hard decision of whether to reduce according to a (and which?) rule, or to shift the next symbol. This is done by the `select'` function, which is allowed to inspect the first input symbol `x` and the *entire* stack: after all, in needs to find the right hand side of a rule on the stack.
In the right column in figure 11.1 the LR derivation of sentence 1+2*3 is shown. Compare closely to the left column, which shows the LL derivation, and note the duality of the processes.

LR action selection

The choice whether to shift or to reduce is made by function `select'`. It is defined as follows:

```
select' as x
   | null items    = Error
   | null redItems = Shift
   | otherwise     = Reduce a (length rs)
       where items    =  dfa as
             redItems =  filter red items
             (a,rs,_) =  hd redItems
```

In the selection process a set (list) of so-called *items* plays a role. If the set is empty, there is an error. If the set contains at least one item, we filter the `red`, or *reducible* items from it. There should be only one, if the grammar has the LR-property. (Or rather: it is the LR-property that there is only one element in this situation). The reducible item is the production rule that can be reduced by the stack machine.

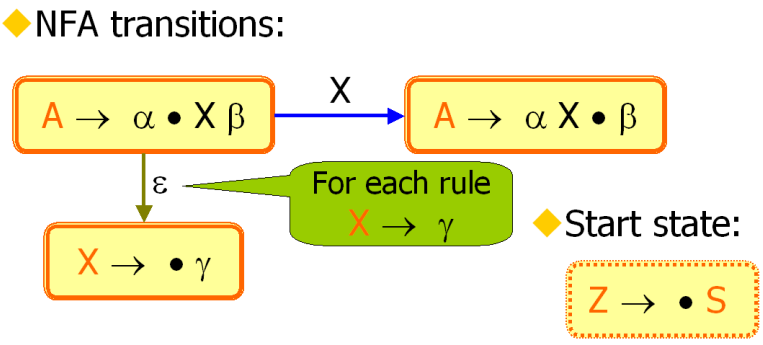
Now what are these items? An *item* is defined to be a production rule, augmented with a 'cursor position' somewhere in the right hand side of the rule. So for the rule $F \rightarrow$ `(`E`)`, there are four possible items: $F \rightarrow \cdot$`(`E`)`, $F \rightarrow$ `(` $\cdot$ E`)`, $F \rightarrow$ `(`$E \cdot$ `)` and $F \rightarrow$ `(`E`)`$\cdot$, where $\cdot$ denotes the cursor.

for the rule $F \rightarrow$ `2` we have two items: one having the cursor in front of the single symbol, and one with the cursor after it: $F \rightarrow \cdot$ `2` and $F \rightarrow$ `2` $\cdot$. For epsilon-rules there is a single item, where the cursor is at position 0 in the right hand side.

In the Haskell representation, the cursor is an integer which is added as a third element of the tupel, which already contains nonterminal and right hand side of the rule.

The items thus constructed are taken to be the states of a NFA (Nondeterministic Finite-state Automaton), as described in section 5.1.2. We have the following transition relations in this NFA:

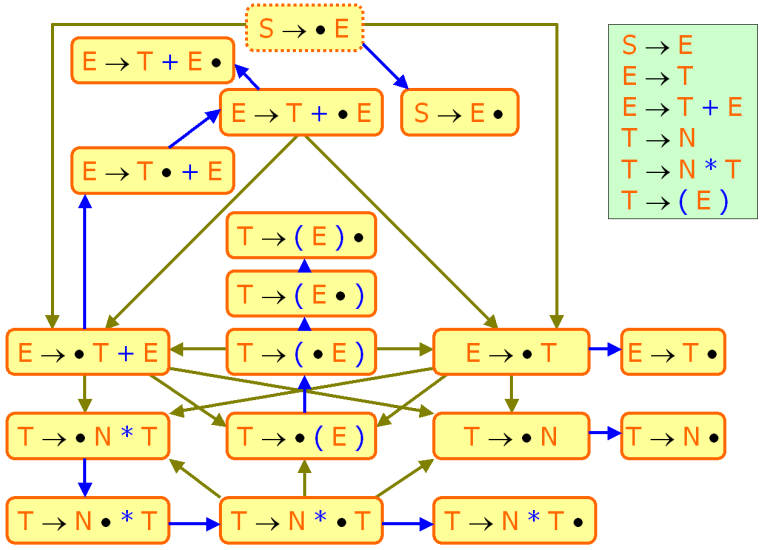
- The cursor can be 'stepped' to the next position. This transition is labeled with the symbol that is hopped over
- If the cursor is on front of a nonterminal, we can jump to an item describing the application of that nonterminal, where the cursor is at the beginning. This relation is an epsilon-transition of the NFA.



As an example, let's consider an simplification of the arithmetic expression grammar:

$$
\begin{aligned}
E &\rightarrow T \\
E &\rightarrow T \texttt{ + } E \\
T &\rightarrow N \\
T &\rightarrow N \texttt{ * } T \\
T &\rightarrow \texttt{(} E \texttt{)}
\end{aligned}
$$

it skips the 'factor' notion as compared to the grammar earlier in this chapter, so it misses some well-formed expressions, but it serves only as an example for creating the states here. There are 18 states in this machine, as depicted here:

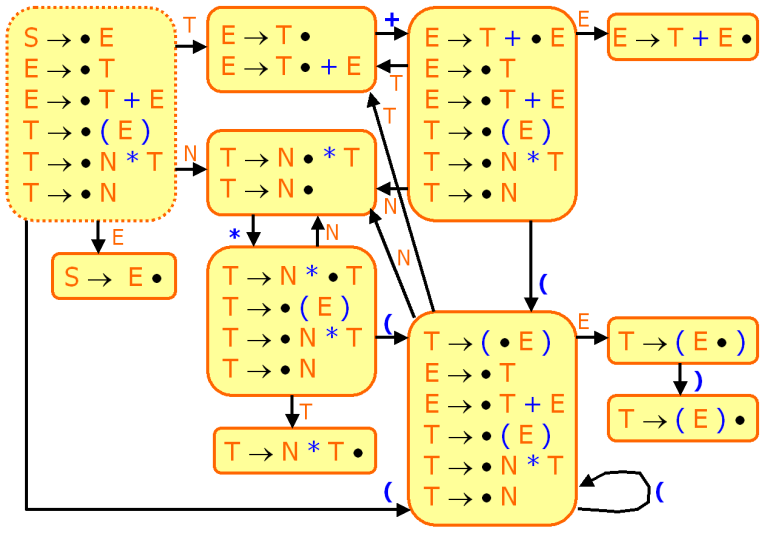


Exercise 11.1 ▷ How can you predict the number of states from inspecting the grammar?
◁

Exercise 11.2 ▷ In the picture, the transition arrow labels are not shown. Add them.
◁

Exercise 11.3 ▷ What makes this FA nondeterministic?
◁

As was shown in section 5.1.4, another automaton can be constructed that is deterministic (a DFA). That construction involves defining states which are *sets* of the original states. So in this case, the states of the DFA are *sets of items* (where items are rules with a cursor position). In the worst case we would have 2^{18} states for the DFA-version of our example NFA, but it turns out that we are in luck: there are only 11 states in the DFA. Its states are rather complicated to depict, as they are sets of items, but it can be done:



Exercise 11.4 ▷ Check that this FA is indeed deterministic.
◁

Given this DFA, let's return to our function that selects whether to shift or to reduce:

```
select' as x
   | null items    = Error
   | null redItems = Shift
   | otherwise     = Reduce a (length rs)
       where items    =  dfa as
             redItems =  filter red items
             (a,rs,_) =  hd redItems
```

It runs the DFA, *using the contents of the stack (*`as`*) as input.* From the state where the DFA ends, which is by construction a set of items, the 'red' ones are filtered out. An item is 'red' (that is: reducible) if the cursor is at the end of the rule.

Exercise 11.5 ▷ Which of the states in the picture of the DFA contain red items? How many? ◁

This is not the only condition that makes an item reducible; the second condition is that the lookahead input symbol is in the *follow* set of the nonterminal that is reduced to. Function `follow` was also needed in the LL analysis in section 10.2.8. Both conditions are in the formal definition:

```
... where red (a,r,c)  =  c==length r   &&  x `elem` follow a
```

Exercise 11.6 ▷ How does this condition reflect that 'the cursor is at the end'? ◁

LR optimizations and generalizations

The stack machine `run'`, by its nature, pushes and pops the stack continuously, and does a recursive call afterwards. In each call, for making the shift/reduce decision, the DFA is run on the (new) stack. In practice, it is considered a waste of time to do the full DFA transitions each time, as most of the stack remains the same after some popping and pushing at the top. Therefore, as an optimization, at each stack position, the corresponding DFA state is also stored. The states of the DFA can easily be numbered, so this amounts to just storing extra integers on the stack, tupled eith the symbols that used to be on the stack. (An artificial bottom element should be placed on the stack initially, containing a dummy symbol and the number of the initial DFA state).

By analysing the grammar, two tables can be precomputed:

- Shift, that decides what is the new state when pushing a terminal symbol on the stack. This basically is the transition relation on the DFA.
- Action, that decides what action to take from a given state seeing a given input symbol.

Both tables can be implemented as a two-dimensional table of integers, of dimensions the number of states (typically, 100s to 1000s) times the number of symbols (typically, under 100).

As said in the introduction, the algorithm described here is a mere *Simple* LR parsing, or SLR(1). Its simplicity is in the reducibility test, which says:

```
... where red (a,r,c)  =  c==length r   &&  x `elem` follow a
```

The *follow* set is a rough approximation of what might follow a given nonterminal. But this set is not dependent of the context in which the nonterminal is used; maybe, in some contexts, the set is smaller. So, the SLR `red` function, may designate items as reducuble, where it actually should not. For some grammars this might lead to a decision to reduce, where it should have done a shift, with a failing parser as a consequence.

An improvement, leading to a wider class of grammars that are allowable, would be to make the *follow* set context dependent. This means that it should vary for each *item* instead of for each *nonterminal*. It leads to a dramatic increase of the number of states in the DFA. And the full power of LR parsing is sledomly needed.

A compromise position is taken by the so-called *LALR* parsers, or *Look Ahead LR* parsing. (A rather silly name, as *all* parsers look ahead...). In LALR parsers, the follow sets are context dependent, but when states in the DFA differ only with respect to the follow-part of their set-members (and not with respect to the item-part of them), the states are merged. Grammars that do not give rise to shift/reduce conflicts in this situation are said to be LALR-grammars. It is not really a natural notion; rather, it is a performance hack that gets most of LR power while keeping the size of the goto- and action-tables reasonable.

A widely used parser generator tool named *yacc* (for 'yet another compiler compiler') is based on an LALR engine. It comes with Unix, and was originally created for implementing the first

C compilers. A commonly used clone of yacc is named *Bison*. Yacc is designed for doing the context-free aspects of analysing a language. The micro structure of identifiers, numbers, keywords, comments etc. is not handled by this grammar. Instead, it is described by regular expressions, which are analysed by a accompagnying tool to yacc named *lex* (for 'lexical scanner'). Lex is a preprocessor to yacc, that subdivides the input character stream into a stream of meaningful *tokens*, such as numbers, identifiers, operators, keywords etc.

11.3 Using Lex and Yacc

Let's write a mini calculator. The grammar needed for such a program is quite simple. We will have to evaluate expressions that can be either a number, an expression between parenthesis, an expression added to an other, etc. So, as you can see, the grammar is recursive.
The analysis of the language is made in two passes. The first one is what is called lexical analysis. This is the job of Lex, through the function `yylex()`, which consumes the tokens (see below). This function will inform the syntactical analyser, generated by Yacc, through the function `yyparse()`.

Use of Lex for lexical analysis
The purpose of the lexical analysis is to transform a series of symbols into tokens (a token may be a number, a '+' sign, a reserved word for the language, etc.). Once this transformation is done, the syntactical analyser will be able to do its job (see below). So, the aim of the lexical analyser is to consume symbols and to pass them back to the syntactical analyser.
A Lex description file can be divided into three parts, using the following plan:

```
declarations
%%
productions
%%
additional code
```

in which no part is required. However, the first `%%` is required, in order to mark the separation between the declarations and the productions.

First part of a Lex file: declarations

This part of a Lex file may contain:

- Code written in the target language (usually C or C++), embraced between `%{` and `%}`, which will be placed at the top of the file that Lex will create. That is the place where we usually put the include files. Lex will put 'as is' all the content enclosed between these signs in the target file. The two signs will have to be placed at the beginning of the line.
- Regular expressions, defining non-terminal notions, such as letters, digits, and numbers. This specifications have the form: `notion regular_expression`. You will be able to use the notions defined this way in the end of the first part of the file, and in the second part of the file, if you embrace them between `{` and `}`.

Example:

```
%{
#include "calc.h"
#include <stdio.h>
#include <stdlib.h>
%}
/* Regular expressions */
white       [\t\n ]+
letter      [A-Za-z]
digit10     [0-9]           /* base 10 */
digit16     [0-9A-Fa-f]     /* base 16 */
identifier  {letter}(_|{letter}|{digit10})*
int10       {digit10}+
```

The example by itself is, I hope, easy to understand. Allowable regular expressions for Lex are summarized in figure 12.2.
So the definition:

```
identifier  {letter}(_|{letter}|{digit10})*
```

Symbol	Meaning
`x`	the `x` character
`.`	any character except `\n`
`[xyz]`	either x, y or z
`[^bz]`	any character, except b and z
`[a-z]`	any character between a and z
`[^a-z]`	any character, except those between a and z
`R*`	zero R or more; R can be any regular expression
`R+`	one R or more
`R?`	one or zero R (that is an optional R)
`R{2,5}`	2 to 5 R
`R{2,}`	2 R or more
`R{2}`	exactly 2 R
`"[xyz\"foo"`	the string `"[xyz"foo"`
`{NOTION}`	expansion of NOTION, that as been defined above in the file
`\X`	if X is a `a`, `b`, `f`, `n`, `r`, `t`, or `v`, this represents the ANSI-C interpretation of `\X`
`\0`	ASCII 0 character
`\123`	the caracter which ASCII code is `123`, in octal
`\x2A`	the caracter which ASCII code is `2A`, in hexadecimal
`RS`	R followed by S
`R|S`	R or S
`R/S`	R, only if followed by S
`^R`	R, only at the beginning of a line
`R$`	R, only at the end of a line
`<<EOF>>`	end of file

Figure 11.2: Regular expressions in Lex

will match as identifiers the words `integer`, `a_variable`, `a1`, but not `_ident` nor `1variable`. Easy, isn't it?
As a last example, this is the definition of a real number:

```
digit       [0-9]
integer     {digit}+
exponent    [eE][+-]?{integer}
real        {integer}("."{integer})?{exponent}?
```

Second part of a Lex file: productions

This part is aimed to instruct Lex what to do in he generated analyser when it will encounter one notion or another one. It may contain:

- Some specifications, written in the target language (usually C or C++) surrounded by `%{` and `%}` (at the beginning of a line). The specifications will be put at the beginning of the `yylex()` function, which is the function that consumes the tokens, and returns an integer.
- Productions, having the syntax: `regular_expression action`. If action is missing, Lex will put the matching characters as is into the standard output. If action is specified, it has to be written in the target language. If it contains more than one instruction or is written in more than one line, you will have to embrace it between `{` and `}`. You should also note that comment such as `/* ... */` can be present in the second part of a Lex file only if enclosed between braces, in the action part of the statements. Otherwise, Lex would consider them as regular expressions or actions, which would give errors, or, at least, a weird behaviour.

Finally, the `yytext` variable used in the actions contains the characters accepted by the regular expression. This is a `char` table, of length `yyleng` (i.e., `char yytext[yyleng]`).
Example:

```
%%
[ \t]+$         ;
[ \t]+          printf(" ");
```

This little Lex file will generate a program that will suppress the space characters that are not useful. You can also notice with that little program that Lex is not reserved to interpreters or compilers, and can be used, for example, for searches and replaces, etc.

Third part: additional code

You can put in this optional part all the code you want. If you don't put anything here, Lex will consider that it is just:

```
main() {
  yylex();
}
```

As you can see it, Lex is quite easy to use (but it can be more complicated if you use start conditions). We have not seen all the possibilities of Lex, and I suggest that you read its man page for a more detailed information.

Syntactical analysis with Yacc

Yacc (Yet Another Compiler Compiler) is a program designed to compile a LALR(1) grammar and to produce the source code of the syntactical analyser of the language produced by this grammar. It is also possible to make it do semantic actions.
As for a Lex file, a Yacc file can be divided into three parts:

```
declarations
%%
productions
%%
additional code
```

and only the first `%%` and the second part are mandatory

The first part of a Yacc file

The first part of a Yacc file may contain:

- Specifications written in the target language, enclosed between `%{` and `%}` (each symbol at the begining of a line) that will be put at the top of the scanner generated by Yacc.
- Declaration of the tokens that can be encontered, with the `%token` keyword.
- The type of the terminal, using the reserved word: `%union`
- Informations about operators' priority or associativity.
- The axiom of the grammar, using the reserved word `%start` (if not specified, the axiom is the first production of the second part of the file).
- The `yylval` variable, implicitely declared of the `%union` type is really important in the file, since it is the variable that contains the description of the last token read.

Second part of a Yacc file

This part can not be empty. It may contain:

- Declarations and/or definitions enclosed between `%{` and `%}`
- Productions of the language's grammar.

These productions look like:

```
nonterminal_notion:
                    body_1       { semantical_action_1 }
                  | body_2       { semantical_action_2 }
                  | ...
                  | body_n       { semantical_action_n }
                  ;
```

provided that the `body_i` are terminal or nonterminal notions of the language.

Third part of a Yacc file

This part contains the additional code, must contain a main() function (that should call the `yyparse()` function), and an `yyerror(char *message)` function, that is called when a syntax error is found.

```
%{
#include "global.h"
#include "calc.h"
#include <stdlib.h>
%}

white           [ \t]+
digit           [0-9]
integer         {digit}+
exponent        [eE][+-]?{integer}
real            {integer}("."{integer})?{exponent}?

%%

{white}         { /* We ignore white characters */ }
{real}          { yylval=atof(yytext);
                  return(NUMBER);
                }

"+"   return(PLUS);
"-"   return(MINUS);
"*"   return(TIMES);
"/"   return(DIVIDE);
"^"   return(POWER);
"("   return(LEFT_PARENTHESIS);
")"   return(RIGHT_PARENTHESIS);
"\n"  return(END);
```

Listing 1: demo.lex

This presentation is far from being exhaustive, and I didn't explain you everything. We will clarify some points in the following example.

Example: a little expression interpreter
This interpreter is able to calculate the value of any mathematical expression written with real numbers, the operators `+`, `-` (binary and unary operators), `*`, `/`, `^` (power operator). This interpreter will not be able to use variables or functions.

The Lex part of the interpreter

The Lex part of the parser is shown in listing 1.
Explanation: The first part includes the file `calc.h`, that will be generated later by Yacc and that will contain the definition for `NUMBER`, `PLUS`, `MINUS`, etc. We include the `stdlib` header, because we will use the `atof()` function later. We declare the real notion used in the second part. The `global.h` file contains only the `#define YYSTYPE double` declaration, because all the structures we will manipulate have the type `double`. By the way, this is the type of `yylval`.
The second part tells the syntactical parser which type of token it encountered. If it is a number, we put its value in the `yylval` variable, in order to be used later.
Finally, the third part is empty, because we do not want Lex to create a `main()` function, as that will be declared in the Yacc file.

The Yacc part of the interpreter

This is the most important, and the most interesting. It is shown in listing 2.
Well, it seems a little less simple, isn't it? In fact, it is not so complicated. We include the usual files, and we use the `%token` keyword to declare the tokens that we can find. There is, in this case, no particular order for the declaration.

```
%{
#include "global.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
%}

%token  NUMBER
%token  PLUS    MINUS   TIMES   DIVIDE  POWER
%token  LEFT_PARENTHESIS    RIGHT_PARENTHESIS
%token  END

%left   PLUS    MINUS
%left   TIMES   DIVIDE
%left   NEG
%right  POWER

%start Input
%%

Input:
      /* Empty */
    | Input Line
    ;

Line:
      END
    | Expression END        { printf("Result: %f\n",$1); }
    ;

Expression:
      NUMBER                        { $$=$1;    }
    | Expression PLUS Expression    { $$=$1+$3; }
    | Expression MINUS Expression   { $$=$1-$3; }
    | Expression TIMES Expression   { $$=$1*$3; }
    | Expression DIVIDE Expression  { $$=$1/$3; }
    | MINUS Expression %prec NEG    { $$=-$2;   }
    | Expression POWER Expression   { $$=pow($1,$3); }
    | LEFT_PARENTHESIS Expression RIGHT_PARENTHESIS { $$=$2; }
    ;

%%

int yyerror(char *s) {
  printf("%s\n",s);
}

int main(void) {
  yyparse();
}
```

Listing 2: demo.y

Then we have the `%left` and `%right` keywords. This is used to tell Yacc the associativity of the operators, and their priority. Then, we define the operators in an increasing order of priority. So, `1+2*3` is evaluated as 1+(2*3)". You will have to choose between `%left` or `%right` declaration for your operator. For a left-associative operator (`+` in this example), `a+b+c` will be evaluated as `(a+b)+c`. For a right-associative operator (`^` here), `a^b^c` will be evaluated as `a^(b^c)`.
Then we tell Yacc that the axiom will be `Input`, that is its state will be such as it will consider any entry as an `Input`, at the begining. You should also note the recursion in the definition of `Input`. It is used to treat an entry which size is unknown. For internal reason to Yacc, you should use:

```
Input:
       /* Empty */
     | Input Line
     ;
```

instead of:

```
Input:
       /* Empty */
     | Line Input
     ;
```

(This permits a reduction as soon as possible).
Let's have a look to the definition of `Line`. The definition itself is quite simple, but you should ask yourself what the `$1` represents. In fact, `$1` is a reference to the value returned to the first notion of the production. It is similar for `$2`, `$3`, etc. And `$$` is the value returned by the production. So, in the definition of Expression, `$$=$1+$3` adds the value of the first expression to the value of the second expression (this is the third notion) and returns the result in `$$`.
If you have a look at the definition of the unary minus, the `%prec` keyword is used to tell Yacc that the priority is that of `NEG`.
Finally, the third part of the file is simple, since it just calls the `yyparse()` function.
The output of yacc is a file named `y.tab.c`. It contains the generated code, among which is a definition of the goto- and action-tables. The part of it containing these tables is shown in listing 3.

Compiling and running the example

Provided that the Lex file is called `calc.lex`, and the Yacc file `calc.y`, all you have to do is:

```
bison -d calc.y
mv calc.tab.h calc.h
mv calc.tab.c calc.y.c
flex calc.lex
mv lex.yy.c calc.lex.c
gcc -c calc.lex.c -o calc.lex.o
gcc -c calc.y.c -o calc.y.o
gcc -o calc calc.lex.o calc.y.o -ll -lm [and maybe -lfl]
```

Please note that you need to create a file called `global.h` which will contain:

```
#define YYSTYPE double
extern YYSTYPE yylval;
```

I only have bison and flex instead of yacc and lex, but it should be the same, except the file names. The call to bison with the `-d` parameter creates the `calc.tab.h` header file, which defines the tokens. We call flex and we rename the files we get. Then, you only have to compile, and do not forget the proper libraries. We get:

```
$ calc
 1+2*3
 Result : 7.000000
 2.5*(3.2-4.1^2)
 Result : -34.025000
```

A better calculator

When there is a syntax error, the program stops. In order to continue, we may replace

```
Line:
          END
        | Expression END                { printf("Result : %f\n",$1); }
        ;
```

by

```
Line:
          END
        | Expression END                { printf("Result : %f\n",$1); }
        | error END                     { yyerrok; }
        ;
```

but, of course, this is only an idea and there are many others (usage and definition of variables and functions, many data types, etc.).

Bibliographical notes

The example DFA and NFA for LR parsing, and part of the description of the algorithm were taken from course notes by Alex Aiken and George Necula, which can be found at `www.cs.wright.edu/~tkprasad/courses/cs780`

Section 12.3 on Lex and Yacc was written by Etienne Bernard, and can be found at `pltplp.net/lex-yacc`.

```
static YYCONST yytabelem yyexca[] ={
-1, 1,
0, -1,
-2, 0,
};
# define YYNPROD 13
# define YYLAST 35
static YYCONST yytabelem yyact[]={
     9,    10,    11,    12,    13,    13,     2,     8,     9,    10,
    11,    12,    13,     5,    21,     6,     5,     1,     6,     7,
     4,     3,     7,    11,    12,    13,     0,    14,    15,     0,
    16,    17,    18,    19,    20 };
static YYCONST yytabelem yypact[]={
-10000000,  -244,-10000000,-10000000,  -258,-10000000,  -241,  -241,-10000000,  -241,
  -241,  -241,  -241,  -241,  -257,  -250,  -237,  -237,  -257,  -257,
  -257,-10000000 };
static YYCONST yytabelem yypgo[]={
     0,    17,     6,    20 };
static YYCONST yytabelem yyr1[]={
     0,     1,     1,     2,     2,     3,     3,     3,     3,     3,
     3,     3,     3 };
static YYCONST yytabelem yyr2[]={
     0,     0,     4,     2,     5,     3,     7,     7,     7,     7,
     5,     7,     7 };
static YYCONST yytabelem yychk[]={
-10000000,    -1,    -2,   265,    -3,   257,   259,   263,   265,   258,
   259,   260,   261,   262,    -3,    -3,    -3,    -3,    -3,    -3,
    -3,   264 };
static YYCONST yytabelem yydef[]={
     1,    -2,     2,     3,     0,     5,     0,     0,     4,     0,
     0,     0,     0,     0,    10,     0,     6,     7,     8,     9,
    11,    12 };
typedef struct
yytoktype
{ char *t_name; int t_val; } yytoktype;

yytoktype yytoks[] =
{
"NUMBER", 257,
"PLUS", 258,
"MINUS", 259,
"TIMES", 260,
"DIVIDE", 261,
"POWER", 262,
"LEFT_PARENTHESIS", 263,
"RIGHT_PARENTHESIS", 264,
"END", 265,
"NEG", 266,
"-unknown-", -1 /* ends search */
};
```

Listing 3: y.tab.c, part 1 of 1