Syntax Analysis (continued)

Bottom-up Parsing

In the previous section on top-down parsers we saw that these parsers start with the start symbol of a context-free grammar and build parse trees going down. In this section on bottom-up parsers we are going to see that these parsers start from the bottom of the parse trees and build them going up towards the start symbol of a context-free grammar.

In the previous section we saw that top-down parsers compute a leftmost derivation of the input string. In this section we are going to see that bottom-up parsers compute a rightmost derivation of the input string. There is going to be a problem with this though: to compute a rightmost derivation, a parser would have to read the input from right to left, not left to right. Since bottom-up parsers actually read input from left to right, what bottom-up parsers compute is in fact a *rightmost derivation in reverse*.

In the previous section we saw that top-down parsers use a stack (either an implicit stack of function calls for recursive-descent parsers and (recursive) predictive parsers, or an explicit stack for nonrecursive predictive parsers) to describe what they expect to see at various points in the remaining input. In particular the top of the stack is used to indicate what the parsers expect to see next in the input. In this section we are going to see that bottom-up parsers use a stack to describe what they have seen at various points in the input already consumed. In particular the top of the stack will be used to indicate what was seen last in the input. To do this, bottom-up parsers always move tokens from the input to the top of the stack as they are processed by the parser. This is called a *shift* from the input to the stack.

In the previous section we saw that a top-down parser use grammars productions like $X \to UVW$ to replace a more abstract X at the top of the stack with the more concrete UVW. By starting from the start symbol of the grammar and repeatedly doing such stack replacements, the parser expends the stack step by step until all the input has been matched and the stack is empty. In this section we are going to see that a bottom-up parser uses grammars productions like $X \to UVW$ to replace a more concrete UVW at the top of the stack with the more abstract X. By starting from an empty stack and repeatedly doing shifts and such stack replacements, the parser contracts the stack step by step until all the input has been matched and only the start symbol of the grammar remains at the top of the stack. Contracting the stack like this is called doing

a *reduce* of the stack.

The two previous paragraphs then explain why bottom-up parsers are often called *shift-reduce* parsers. There is no equivalent name for top-down parsers...

Here is a small example to explain how a bottom-up (shift-reduce) parser works. Consider the following grammar:

$$\begin{array}{lcl} S & \rightarrow & \texttt{a}A B\texttt{e} \\ A & \rightarrow & A\texttt{bc} \\ A & \rightarrow & \texttt{b} \\ B & \rightarrow & \texttt{d} \end{array}$$

To parse the input string `abbcbcde`, the bottom-up parser starts with an empty stack:

Stack	Input	Output
	`abbcbcde`	

The first thing a bottom-up parser always does is then a shift of the first input token onto the stack:

Stack	Input	Output
`a`	`bbcbcde`	shift

There is no production that the parser can use at this point to reduce the content of the stack, so the parse shifts the next input token onto the stack:

Stack	Input	Output
`ab`	`bcbcde`	shift

The parser can now use the grammar production $A \rightarrow$ `b` to reduce the content of the stack:

Stack	Input	Output
aA	`bcbcde`	reduce using $A \rightarrow$ `b`

Note how the grammar production is used "backwards", going from the more precise RHS of the production back to the more abstract LHS. This is opposite the way we used grammar productions in (nonrecursive predictive) top-down parsers. At this point in the parsing process, the stack tells us that the input we have processed so far was a token `a` followed by some other input that had the structure of an A, without telling us what that other input was.

The stack cannot be reduced more than that at this point, so the parser shifts another token from the input:

Stack	Input	Output
aA`b`	`cbcde`	shift

At this point the parser could use the grammar production $A \rightarrow$ `b` to reduce the stack again, just as it did two steps ago. In fact the parser will *not* do this and will shift the next input token onto the stack instead. The reason

the parser shifts instead of reducing is because the parser has to compute a rightmost derivation (in reverse) of the input string and we will see below that, if the parser were to do a reduce instead of a shift at this step, then the resulting derivation would not be rightmost (in fact the parser would not be able to finish parsing the input at all). Another way to look at it is to say that, if the parser were to do a reduce instead of a shift at this step then the stack would contain `aAA`, but the grammar above does not allow two `A` nonterminals to appear one after the other, so doing a shift instead of a reduce is the right thing to do at this step. Later when we look at how to construct bottom-up parsers we will see that deciding when to shift and when to reduce is in fact what makes constructing bottom-up parsers difficult.

Stack	Input	Output
aA`bc`	`bcde`	shift

At this point the parser can use the production $A \to A\texttt{bc}$ to reduce the stack:

Stack	Input	Output
aA	`bcde`	reduce using $A \to A\texttt{bc}$

Here the stack tells us again that the input we have processed so far was a token `a` followed by some other input that had the structure of an A, without telling us what that other input was. As we indicated above, the stack always describes what the bottom-up parser has seen at various points in the input already consumed, but we can see here that in fact the parser only keeps an abstract summary of that information on its stack.

After that once again the parser has to shift a token:

Stack	Input	Output
aA`d`	`e`	shift

which can then be reduced using the grammar production $B \to \texttt{d}$:

Stack	Input	Output
aAB	`e`	reduce using $B \to \texttt{d}$

The parser then shifts the last input token:

Stack	Input	Output
aAB`e`		shift

and can reduce the whole stack into the start symbol of the grammar:

Stack	Input	Output
S		reduce using $S \to \texttt{a}AB\texttt{e}$

At this point the input is empty and the stack only contains the start symbol of the grammar. This means that the input string has a structure that follows

the structure of a whole complete program (since the start symbol of a grammar represents all the strings that form complete programs) and the parser accepts the input string.

Here is the complete result of the parsing process:

Stack	Input	Output
	abbcbcde	
a	bbcbcde	shift
ab	bcbcde	shift
aA	bcbcde	reduce using $A \rightarrow$ b
aAb	cbcde	shift
aAbc	bcde	shift
aA	bcde	reduce using $A \rightarrow A$bc
aAb	cde	shift
aAbc	de	shift
aA	de	reduce using $A \rightarrow A$bc
aAd	e	shift
aAB	e	reduce using $B \rightarrow$ d
aABe		shift
S		reduce using $S \rightarrow$ aABe

Note how shifting tokens grows the stack, and how using grammar productions to do reductions shrinks the size of the stack. This is the opposite of what happened with top-down parsers, where grammar productions were used to grow the stack and the stack was shrunk by matching tokens at the top of the stack with tokens in the input.

Here is then the list of grammar productions used by the parser in all the "reduce" steps of the parsing process just above:

$$A \rightarrow \texttt{b}$$
$$A \rightarrow A\texttt{bc}$$
$$A \rightarrow A\texttt{bc}$$
$$B \rightarrow \texttt{d}$$
$$S \rightarrow \texttt{a}AB\texttt{e}$$

If we now read this list *backwards* (from the last line back to the first one) we get the list of grammar productions to use in a rightmost derivation of the input sentence:

	S	start symbol
$\Rightarrow$	aABe	using $S \rightarrow$ aABe
$\Rightarrow$	aAde	using $B \rightarrow$ d
$\Rightarrow$	aAbcde	using $A \rightarrow A$bc
$\Rightarrow$	aAbcbcde	using $A \rightarrow A$bc
$\Rightarrow$	abbcbcde	using $A \rightarrow$ b

Note that the leftmost b in the sentence is the direct result of the last derivation step, the one that uses the grammar production $A \rightarrow$ b. The second

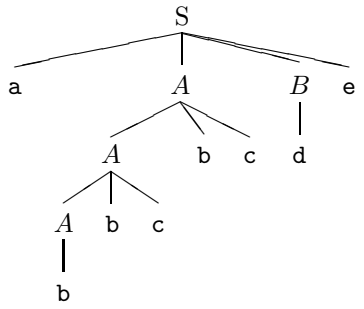
leftmost `b`, on the other hand, appears as part of the result of the previous derivation step, the one that uses the grammar production $A \rightarrow A\texttt{bc}$. This explains why the parser did not reduce the second input token `b` into an A immediately after shifting it onto the stack towards the beginning of the parsing process above: if the parser had reduce that second `b` into an A instead of shifting the `c` that followed it, then the resulting list of grammar productions would not have corresponded to a rightmost derivation of the input sentence (in fact it would not have been possible to finish parsing the input...) In general, even if what appears at the top of the stack matches the RHS of a grammar production, the parser should reduce the stack using that grammar production *only* when that RHS appears as the result of a derivation step in the rightmost derivation of the input sentence.

The question is then: how does a bottom-up parser recognize when to do a shift (like in the case of the second `b` in the input) and when to do a reduce (like in the case of the first `b` in the input)? How does the parser decide which step is part of a rightmost derivation and which step is not?

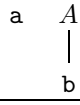
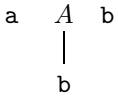
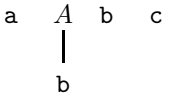
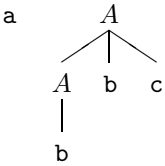
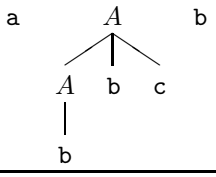
In the example above, the parser can answer that question by looking at both the content of the stack from top to bottom and at the next token in the input: if what is in the stack below the `b` is an `a` and the next input token is a `b` then the parser should reduce the `b` at the top of the stack into an A; if what is in the stack below the `b` is `a`A and the next token is a `c` then the parser should not reduce the `b` at the top and should instead shift the next token `c`. Remember that the parser uses the stack to always keep track of what it has seen so far in the input, so the parser can use this information and the next token in the input to find out which situation it is currently in and decide whether a shift or a reduce is required next.

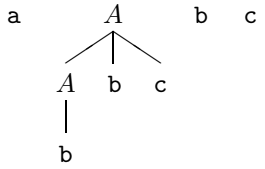
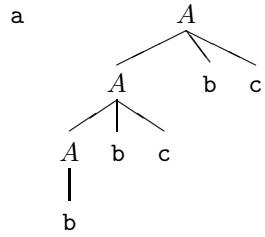
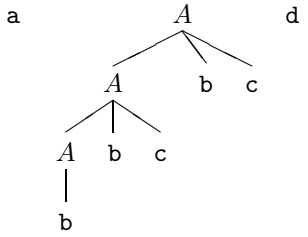
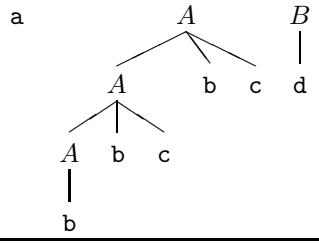
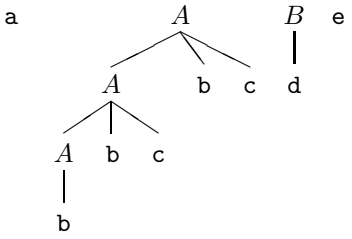
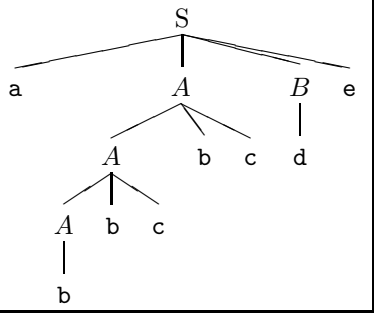
Looking at the content of the stack from top to bottom at each step of the parsing process is slow though, so what the parser will in fact do is keep on the stack extra information about which situation the parser is currently in. This extra information will be in the form of state numbers, with each state representing one possible situation for the parser. The parser will then only need to look at the current state at the top of the stack and the next input token to decide what to do next. We will see below that these different states correspond in fact to the states of a DFA that recognizes when to reduce the stack by matching the content of the stack with the RHS of grammar productions.

Based on the rightmost derivation above the following parse tree can then be built:



In fact if you compare the result of the parsing process with the way the parse tree is built, you can see that each shift corresponds to adding a new leaf to the parse tree, starting with the leftmost leaf and going towards the rightmost leaf (since the input string is read from left to right), and each reduce corresponds to adding a new interior node on top of the rightmost pieces of the parse tree, until the root of the parse tree is built:

Stack	Input	Output	Parse tree
	abbcbcde		
a	bbcbcde	shift	
ab	bcbcde	shift	
aA	bcbcde	reduce using $A \to$ b	
aAb	cbcde	shift	
aAbc	bcde	shift	
aA	bcde	reduce using $A \to A$bc	
aAb	cde	shift	

Stack	Input	Output	Parse tree
aAbc	de	shift	
aA	de	reduce using $A \to A$bc	
aAd	e	shift	
aAB	e	reduce using $B \to$ d	
aABe		shift	
S		reduce using $S \to$ aABe	

As you can see the parse tree is built starting from the leaves going up, which explains why such parsers are called bottom-up parsers.

Here is another example, using a grammar for arithmetic expressions:

$$\begin{array}{lcl} E & \rightarrow & E \texttt{ + } E \\ E & \rightarrow & E \texttt{ * } E \\ E & \rightarrow & \texttt{(} E \texttt{)} \\ E & \rightarrow & \texttt{id} \end{array}$$

This grammar is obviously ambiguous, because it does not specify the associativity of addition, the associativity of multiplication, and the precedence between addition and multiplication.

Note that this grammar is also not left-factored and has two productions which are left-recursive. Fortunately bottom-up parsers do not care whether a grammar is left-factored or not, or whether a grammar is left-recursive or not, so we will not have to worry about these things (this means that bottom-up parsers are actually more powerful than top-down parsers, as bottom-up parsers can deal with more kinds of grammars than top-down parsers).

Given the input string `id + id * id` we will have the following parsing process:

Stack	Input	Output
	`id + id * id`	
`id`	`+ id * id`	shift
E	`+ id * id`	reduce using $E \rightarrow$ `id`
E `+`	`id * id`	shift
E `+ id`	`* id`	shift
E `+` E	`* id`	reduce using $E \rightarrow$ `id`

At this point in the parsing process, the parser has two choices:

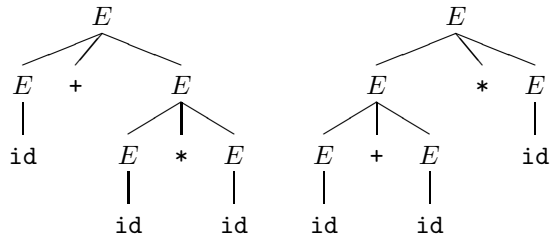
- shift the next `*` token from the input:

Stack	Input	Output
E `+` E `*`	`id`	shift
E `+` E `* id`		shift
E `+` E `*` E		reduce using $E \rightarrow$ `id`
E `+` E		reduce using $E \rightarrow E$ `*` E
E		reduce using $E \rightarrow E$ `+` E

- reduce what is already on the stack using the grammar production $E \rightarrow E$ `+` E:

Stack	Input	Output
E	`* id`	reduce using $E \rightarrow E$ `+` E
E `*`	`id`	shift
E `* id`		shift
E `*` E		reduce using $E \rightarrow$ `id`
E		reduce using $E \rightarrow E$ `*` E

This is called a *shift-reduce conflict*. Doing a shift corresponds to `*` having higher precedence than `+`, while doing a reduce corresponds to `*` having lower precedence than `+`. Obviously the two choices lead to two different parse trees:



The cause of the shift-reduce conflict is obvious here: the grammar we are using is ambiguous, so more than one rightmost derivation of the input string is possible and the parser has no way to decide which one to compute. We will see later though that there are unambiguous grammars with shift-reduce conflicts as well, because bottom-up parsers cannot handle all unambiguous grammars (though we will see that in practice they are able to handle most of the unambiguous grammars we care about).

When a bottom-up parser encounters a shift-reduce conflict like this, it could potentially try one of the two possibilities and, if that does not work, backtrack and try the other possibility. In practice though parsers do not bother with backtracking because it is complicated to implement and not very efficient. A parser will instead always prefer one kind of action over the other and report a syntax error if that choice proves to be the wrong one. For example, parsers generated using YACC will always prefer to shift the next token over reducing the stack when faced with a shift-reduce conflict. If that turns out to be the wrong choice (and the parser has no way to know this in advance) then the parser will simply raise a syntax error and switch to error recovery.

Tools like YACC can always immediately tell the user when a grammar specification (a grammar in a "`.y`" file) leads to a shift-reduce conflict though. In addition the default action of shifting is often the right thing to do in practice, and tools like YACC also provide simple ways to force a reduce to happen instead of a shift if a reduce is required, by allowing the user to add special associativity and precedence information to the grammar without changing it. So shift-reduce conflicts are usually easy to solve without having to rewrite the grammar.

Here is another common example of a shift-reduce conflict with a grammar for `if` statements:

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{if (} E \texttt{)} \ S \\
S & \rightarrow & \texttt{if (} E \texttt{)} \ S \ \texttt{else} \ S \\
S & \rightarrow & \texttt{other}
\end{array}
$$

When given an input sentence with the following general structure (we are not showing all the tokens for E_1, E_2, S_1, and S_2 here to save space):

$$\texttt{if (} E_1 \texttt{)} \ \texttt{if (} E_2 \texttt{)} \ S_1 \ \texttt{else} \ S_2$$

the parser will shift and reduce as usual until it reaches this point in the parsing process:

Stack	Input	Output
`if (` E_1 `) if (` E_2 `)` S_1	`else` S_2	

The parser then has the choice between shifting the "`else` S_2" from the input to the stack followed by reducing this `else` part together with the topmost `if` in the stack, or reducing the topmost `if` in the stack first followed by shifting the "`else` S_2" from the input to the stack to be reduced later together with the bottommost `if`. In the first case the `else` part is then associated with the closest (topmost in the stack) `if` (which is what normally happens in real programming languages) while in the second case the `else` part is associated with the `if` that is the furthest away (bottommost in the stack). Again the problem here is that the grammar is ambiguous. Rewriting the grammar to force the `else` part to always be associated with the closest `if` then removes the shift-reduce conflict.

Finally, here is an example of a *reduce-reduce conflict*, from the Ada language:

$$\begin{array}{lcl} F & \rightarrow & \texttt{id (}\ L\ \texttt{)} \\ A & \rightarrow & \texttt{id (}\ L\ \texttt{)} \\ L & \rightarrow & \varepsilon \mid \texttt{id}\ T \\ T & \rightarrow & \texttt{,}\ \texttt{id}\ T \mid \varepsilon \end{array}$$

In this grammar L represents a (possibly empty) list of identifiers (like `x,y,z`) with T representing the tail of such a list (like `,y,z`). The nonterminal F represents a function call (like `foo(x,y,z)`) where `id` is the name of the function to be called. The nonterminal A represents an array access (like `a(x,y,z)`) where `id` is the name of the array to be dereferenced (in the C programming language the same array dereference would be written as `a[x][y][z]`). In turns out that in Ada function calls and array dereferences use the exact same syntax. This means that when a parser tries to parse input like `id(id,id,id)` the parser will shift the tokens onto the stack, reduce `id,id,id` into an L, and then the parser will not know whether to reduce `id(`L`)` into an F using the first grammar production, or reduce `id(`L`)` into an A using the second grammar production. Hence a reduce-reduce conflict.

Reduce-reduce conflicts are usually much harder to solve than shift-reduce conflicts. A shift-reduce conflict can usually be solved by changing the grammar without changing the language described by the grammar (for example, rewriting the grammar for `if` to make it unambiguous, as we have seen before) or simply by adding a few extra annotations about precedence and associativity to the grammar, while a reduce-reduce conflict can usually be resolved only by changing both the grammar and the language (i.e. changing the syntax of array references to look different from function calls) or by doing nasty tricks with the lexer or parser (for example, modifying the parser so that it marks function names and array names in the symbol table when seeing function and array definitions, and then modifying the lexer so that it uses the information in the

symbol table to return different tokens for function names and array names, like, say, `if_func` and `id_array`, instead of just using `id` for both).

Just like for shift-reduce conflicts, tools like YACC can always immediately tell the user when a grammar specification (a grammar in a "`.y`" file) leads to a reduce-reduce conflict. The default action of the generated parser is then to always do a reduce using the first of the two grammar productions involved in the conflict. This may or may not be the right choice, so, when faced with a reduce-reduce conflict, the person writing the grammar will usually have to carefully check that the parser actually makes the correct decision, and, if not, then the person will have to manually resolve the problem as described in the previous paragraph.

LR Parsers

There are various kinds of bottom-up parsers, which work with different kinds of context-free grammars. For example there is a recursive version of bottom-up parsers, called recursive-ascent parsers, which are the symmetric of the top-down recursive-descent parsers and work by implementing one function for each nonterminal in a context-free grammar. We will not look at those recursive-ascent parsers though because they are never used in practice.

Among the various kinds of bottom-up parsers, the most important kind in practice is LR(k) parsers (LR parsers, for short), which parse input from Left to right and create a Rightmost derivation (in reverse) of the input, using at most k tokens of lookahead. There are several reasons why LR parsers are the most important kind of bottom-up parsers:

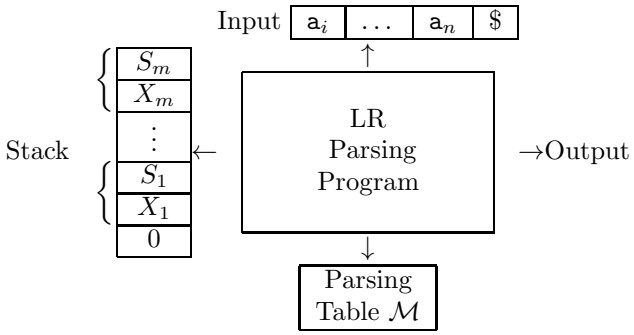
- LR parsers can be used to parse almost all (but not all) programming constructs that can be expressed using a context-free grammar.
- They are the most powerful type of non-backtracking shift-reduce parsers known.
- They can be implemented very efficiently.
- They are strictly more powerful than the top-down predictive parsers we have seen before.

The reason why an LR(k) parser is always more powerful than an LL(k) parser is because the LL(k) parser can look at most at the next k tokens in the input to decide which grammar production to use next to transform the nonterminal at the top of the stack into the corresponding RHS. So in essence an LL(k) can look at most at k tokens of the RHS of grammar productions to make its decision. In the case of the LR(k) parser, on the other hand, the RHS is already completely on the stack at the time a reduction has to be done, and the parser can use both that information plus look at the next k tokens in the input to decide which grammar production to use. So in essence an LR(k) can look at the complete RHS of grammar productions, *plus* the k tokens that

appear in the input *after* the RHS, to make its decision. This means an LR(k) parser has much more information available to make its decision than an LL(k) parser, so the LR(k) parser can handle more complex grammars.

They only disadvantage of LR parsers is that they are complicated to create. So in practice people use tools like YACC or Bison to automatically generate LR parsers from a grammar specification, and only implement by hand the simpler top-down predictive parsers.

The structure of an LR parser is as follows:



Like in a top-down parser, the input is a sequence of tokens with the special symbol \$ at the end. Unlike in a top-down parser, the stack is now not made of just grammar symbols but of pairs of grammars symbols and state numbers. The only exception is the state at the bottom of the stack which is always state 0 and is not part of a pair.

Remember from our introduction to bottom-up parsers above that a parser decides to do a shift or a reduce based on the situation in which it currently is. Since the stack is used by the parser to keep track of what the parser has seen so far in the input, the parser can therefore look at the content of the stack (from top to bottom) at each step of the parsing process, as well as look at the next input token, to decide in which situation it currently is and then use that information to decide whether to do a shift or a reduce next.

Looking at the content of the stack from top to bottom at each step of the parsing process is slow though, so to speed up this process what the parser does is to keep on the stack different state numbers that directly tell the parser in which situation it currently is. The parser will then only need to look at the current state at the top of the stack and the next input token to decide what to do next. For example, in the figure above, state S_m tells the parser that grammar symbols $X_m X_{m-1} \dots X_2 X_1$ are currently below S_m on the stack. State S_{m-1} tells the parser that grammar symbols $X_{m-1} \dots X_2 X_1$ are currently below S_{m-1} on the stack. State S_1 tells the parser that the grammar symbol X_1 is currently below S_1 on the stack. State 0 tells the parser that there are no grammar symbol below that state on the stack. In essence, each state on the stack is a complete summary of all the information below that state in the stack, so the parser only needs to look at the state at the top of the stack to know the whole content of the stack. In turn this means that the parser only needs

to look at the state at the top of the stack and at the next token in the input to know in which situation it currently is with regard to the parsing process.

There are three things to note about these states on the stack:

- We will see below that these different states correspond in fact to the states of a DFA that matches the content of the top of the stack with the RHS of grammar productions. When a match occurs, the parser does the corresponding reduction.
- The states on the stack are not strictly necessary: at each step the parser could simply look at all the grammar symbols in the stack from top to bottom to decide what to do next, but, as we have said above, this would be slow and the resulting parser would therefore be less efficient.
- If the stack contains state numbers then it does not strictly need to contain the associated grammar symbols: as we have just said the states on the stack are enough information for the parser to decide what to do next, so the parser does not actually need to look at the grammar symbols themselves, only at the state numbers. In these lecture notes we will nevertheless show the grammar symbols on the stack to better explain how the parser is using the stack. In practice though, parsers generated by tools like YACC or Bison only use state numbers on their stacks and the grammar symbols are left implicit.

Another difference between top-down and bottom-up parsers is the content of the parsing table $\mathcal{M}$. In a top-down parser the parsing table consists of one row for each nonterminal and one column for each token of lookahead (plus the \$ symbol). In an LR parser the parsing table has two different parts: ACTION and GOTO. The ACTION part of the table has one row for each state and one column for each token of lookahead (plus the \$ symbol), while the GOTO part of the table has one row for each state and one column for each nonterminal. The ACTION part is used by the parser at each step to decide whether to shift or reduce. The GOTO part is used by the parser during a reduction.

Parsing Algorithm for LR Parsers

To describe the LR parsing algorithm, we first need to define *configurations*: a configuration is simply the current content of the stack glued together with the current tokens in the input:

$$\langle 0 \, \underbrace{X_1 S_1} \, \underbrace{X_2 S_2} \, \ldots \underbrace{X_{m-1} S_{m-1}} \, \underbrace{X_m S_m} \;\; \mathtt{a}_i \; \mathtt{a}_{i+1} \; \ldots \mathtt{a}_n \; \$ \rangle$$

Such a configuration completely describes the current state of the parser. The parsing process can then be described as a sequence of steps that change the current state of the parser from one configuration to another.

Note that each configuration directly corresponds to a sentential form in a rightmost derivation of the input:

$$X_1\ X_2\ \ldots\ X_{m-1}\ X_m\ \mathtt{a}_i\ \mathtt{a}_{i+1}\ \ldots\ \mathtt{a}_n$$

This should not be a surprise since bottom-up parsers compute a rightmost derivation (in reverse) of their input. The only difference between a configuration and the corresponding sentential form is the presence in the configuration of the state numbers.

To parse some input using an LR parser, we can then do the following:

- Put the special symbol \$ at the end of the input.
- Put state 0 at the bottom of the stack.
- At each step of the parsing process, if the parser is currently in the configuration $\langle 0\ \underbrace{X_1S_1}\ \underbrace{X_2S_2}\ \ldots \underbrace{X_{m-1}S_{m-1}}\ \underbrace{X_mS_m}\ \mathtt{a}_i\ \mathtt{a}_{i+1}\ \ldots \mathtt{a}_n\ \$\rangle$, the parser considers the state S_m at the top of the stack and the next input symbol $\mathtt{a}_i$ (which can be a token or \$) and consults the parsing table entry ACTION$[S_m, \mathtt{a}_i]$:
  - If ACTION$[S_m, \mathtt{a}_i]$ is "shift S" then the next configuration of the parser is: $\langle 0\ \underbrace{X_1S_1}\ \underbrace{X_2S_2}\ \ldots \underbrace{X_{m-1}S_{m-1}}\ \underbrace{X_mS_m}\ \underbrace{\mathtt{a}_i\ S}\ \mathtt{a}_{i+1}\ \ldots \mathtt{a}_n\ \$\rangle$.
  - If ACTION$[S_m, \mathtt{a}_i]$ is "reduce $A \to \beta$" then the next configuration of the parser is: $\langle 0\ \underbrace{X_1S_1}\ \underbrace{X_2S_2}\ \ldots \underbrace{X_{m-r}S_{m-r}}\ \underbrace{A\ S}\ \mathtt{a}_i\ \mathtt{a}_{i+1}\ \ldots \mathtt{a}_n\ \$\rangle$, where $r = |\beta|$ and $S = \text{GOTO}[S_{m-r}, A]$. The parser outputs the grammar production $A \to \beta$.
  - If ACTION$[S_m, \mathtt{a}_i]$ is "accept" then the parser accepts the input and the current configuration of the parser must be: $\langle 0\ \underbrace{XS_1}\ \$\rangle$, where X is the start symbol of the grammar.
  - If ACTION$[S_m, \mathtt{a}_i]$ is blank then the parser has detected a syntax error and switches to error recovery.

If, while the algorithm is running, ACTION$[S_m, \mathtt{a}_i]$ is "shift S" then the parser moves the symbol $\mathtt{a}_i$ from the input to a new pair at the top of the stack. The state associated with $\mathtt{a}_i$ in the new pair is the state S specified by the shift action.

If, while the algorithm is running, ACTION$[S_m, \mathtt{a}_i]$ is "reduce $A \to \beta$" then the parser first removes from the top of the stack r pairs, where r is the number of grammar symbols in the RHS β of the grammar production to use for the reduction. Note that the r grammar symbols in the r pairs which are removed from the top of the stack are guaranteed by the algorithm to be exactly the same r grammar symbols that appear in β (otherwise it would not make sense to use that grammar production for the reduction). So β is guaranteed to be the same as $X_{m-r+1} \ldots X_m$ (the symbol X_m that was at the top of the stack has to be the rightmost symbol in β, etc.) Once the r pairs have been removed, a new pair is created at the top of the stack. The grammar symbol in that new pair is the nonterminal A from the LHS of the grammar production. This

means that the r grammar symbols that were at the top of the stack have now been abstracted into an A. The parser then uses the GOTO part of the parsing table to compute the state S associated with A in the new pair at the top of the stack.

Remember that every state in the stack is in essence a complete summary of all the information below that state in the stack. This means that the state S_{m-r} which is directly below A in the stack (directly on the left of A in the corresponding configuration) is a summary of the whole stack below it (a summary of everything on its left in the configuration). To compute the state S associated with A in the new pair at the top of the stack, in other words to compute the new summary for the whole stack (including the new pair containing A) the parser takes the topmost summary S_{m-r} and uses the GOTO part of the parsing to combine this summary S_{m-r} with the new nonterminal A at the top of the stack to get the new summary S for the whole stack.

Another way to look at how reductions occur and how the GOTO part of the parsing table is used is to say that, starting from the state S_m that was at the top of the stack, removing the r pairs from the stack brings the parser back into a previous state S_{m-r}, at which point the parser does a transition on A to a new state S. The GOTO part of the parsing table then corresponds to (part of) the transition table of a DFA which has a transition on A going from state S_{m-r} to state S of the DFA. In fact this is exactly how we will later construct parsing tables: by first computing DFAs that describe the states of the parsers and the transitions between states.

Note that, while a reduction occurs, the input tokens do not change. A reduction is purely a manipulation of the parser's stack to reduce its size by transforming a more concrete sequence of symbols β at the top of the stack into a more abstract (less specific) nonterminal A. Remember that the parser always uses the stack to keep track of what it has seen so far in the input. By doing a reduction the parser simply switches to a more abstract version of what it remembers about the structure of the input it has seen so far.

Shifts and reductions continue like this until either the parser reaches a blank entry in the parsing table, in which case a syntax error is raised, or until the parser accepts the input. When the parser accepts the input, the algorithm guarantees that only the special symbol \$ remains in the input (i.e. all the input tokens have been consumed) and the only grammar symbol remaining on the stack is the start symbol of the grammar (meaning that the input sentence has been correctly identified as a complete program). State 0 always remains at the bottom of the stack during the whole parsing process. The state associated with the start symbol of the grammar in the last pair on the stack when the parser accepts can be any state, it does not matter.

Now that we have seen how the algorithm works, we can look at an example. Here is an unambiguous grammar for arithmetic expression, where each production is numbered:

$$
\begin{array}{llll}
1 & E & \rightarrow & E \texttt{ + } T \\
2 & E & \rightarrow & T \\
3 & T & \rightarrow & T \texttt{ * } F \\
4 & T & \rightarrow & F \\
5 & F & \rightarrow & \texttt{(} E \texttt{)} \\
6 & F & \rightarrow & \texttt{id}
\end{array}
$$

Note how this grammar has two left-recursive productions (one for E and one for T) but bottom-up parsers do not care about that (or about grammar productions with common prefixes).

We will see later how to construct a parsing table from this grammar. Here I give you directly the parsing table so we can use it to parse some input:

State	ACTION						GOTO		
	`id`	`+`	`*`	`(`	`)`	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

In this table we use for example "s5" to mean "shift 5" and we use "r5" to mean "reduce using grammar production number 5" ($F \rightarrow$ `(` E `)`). So in "s5" the 5 represents a state number, while in "r5" the 5 represents a grammar production number.

Using this parsing table we can then parse the following input: `id + id * id` as follows. First we put on the stack the state 0. We also put a $ symbol at the end of the input.

Stack	Input	Output
0	`id + id * id` $	

The state at the top of the stack is 0 and the next input symbol is `id`. The table entry ACTION[0, `id`] is "shift 5" so we move the token `id` from the input to a new pair on the stack. The new state associated with `id` in the new pair is 5, as indicated by the shift action:

Stack	Input	Output
0 $\underbrace{\texttt{id}\ 5}$	`+ id * id` $	shift 5

The state at the top of the stack is now 5 and the next input symbol is `+`. The table entry ACTION$[5, \texttt{+}]$ is "reduce 6". This means we should reduce the content of the stack using grammar production number 6, which is $F \rightarrow \texttt{id}$. The RHS of this grammar production is `id`, which is one grammar symbol long, so we remove exactly one pair from the top of the stack (the pair that precisely contains `id`). Then we add a new pair at the top of the stack for the LHS of the grammar production, which is F. To compute the new state that is associated with this F in the new pair at the top of the stack, we use the GOTO part of the parsing table: we take the state 0 that is directly to the left of F on the stack, take the F itself, and use these two pieces of information to find the new state GOTO$[0, F]$ that should go at the top of the stack in the pair for F. In this case the new state is 3:

Stack	Input	Output
$0\ \underbrace{F\ 3}$	`+ id * id` $	reduce $F \rightarrow$ `id`

The state at the top of the stack is now 3 and the next input symbol is still `+`. The table entry ACTION$[3, \texttt{+}]$ is "reduce 4". This means we should reduce the content of the stack using grammar production number 4, which is $T \rightarrow F$. The RHS of this grammar production is F, which is one grammar symbol long, so we remove exactly one pair from the top of the stack (the pair that precisely contains F). Then we add a new pair at the top of the stack for the LHS of the grammar production, which is T. To compute the new state that is associated with this T in the new pair at the top of the stack, we use the GOTO part of the parsing table again: we take the state 0 that is directly to the left of T on the stack, take the T itself, and use these two pieces of information to find the new state GOTO$[0, T]$ that should go at the top of the stack in the pair for T. In this case the new state is 2:

Stack	Input	Output
$0\ \underbrace{T\ 2}$	`+ id * id` $	reduce $T \rightarrow F$

The state at the top of the stack is now 2 and the next input symbol is still `+`. The table entry ACTION$[2, \texttt{+}]$ is "reduce 2". This means we should reduce the content of the stack using grammar production number 2, which is $E \rightarrow T$. The RHS of this grammar production is T, which is one grammar symbol long, so we remove exactly one pair from the top of the stack (the pair that precisely contains T). Then we add a new pair at the top of the stack for the LHS of the grammar production, which is E. To compute the new state that is associated with this E in the new pair at the top of the stack, we use the GOTO part of the parsing table again: we take the state 0 that is directly to the left of E on the stack, take the E itself, and use these two pieces of information to find the new state GOTO$[0, E]$ that should go at the top of the stack in the pair for T. In this case the new state is 1:

Stack	Input	Output
$0\ \underbrace{E\ 1}$	`+ id * id` $	reduce $E \rightarrow T$

What we have done so far is to transform `id` into an F, then into a T, then into an E, which means that, by itself, an `id` token is an expression.

The state at the top of the stack is now 1 and the next input symbol is still `+`. The table entry ACTION$[1, \texttt{+}]$ is "shift 6" so we move the token `+` from the input to a new pair on the stack. The new state associated with `+` in the new pair is 6, as indicated by the shift action:

Stack	Input	Output
$0\ \underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}$	`id * id` $	shift 6

The state at the top of the stack is now 6 and the next input symbol is `id`. The table entry ACTION$[6, \texttt{id}]$ is "shift 5" so we move the token `id` from the input to a new pair on the stack. The new state associated with `id` in the new pair is 5, as indicated by the shift action:

Stack	Input	Output
$0\ \underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{\texttt{id}\ 5}$	`* id` $	shift 5

The state at the top of the stack is now 5 and the next input symbol is `*`. The table entry ACTION$[5, \texttt{*}]$ is "reduce 6", so we reduce the content of the stack using grammar production number 6, which is $F \rightarrow \texttt{id}$. The RHS of this grammar production is `id`, which is one grammar symbol long, so we remove exactly one pair from the top of the stack (the pair that precisely contains `id`). Then we add a new pair at the top of the stack for the LHS of the grammar production, which is F. To compute the new state that is associated with this F in the new pair at the top of the stack, we use the GOTO part of the parsing table again: we take the state 6 that is directly to the left of F on the stack, take the F itself, and use these two pieces of information to find the new state GOTO$[6, F]$ that should go at the top of the stack in the pair for F. In this case the new state is 3:

Stack	Input	Output
$0\ \underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{F\ 3}$	`* id` $	reduce $F \rightarrow \texttt{id}$

As we have explained before, what happens in essence here is that we take the state 6, which summarizes the grammar symbols E`+` that appear on its left, and combine this old summary with the new nonterminal F to get the new state 3, which summarizes the grammar symbols E`+`F that are now on the stack.

The state at the top of the stack is now 3 and the next input symbol is `*`. The table entry ACTION$[3, \texttt{*}]$ is "reduce 4", so we reduce the content of the stack using grammar production number 4, which is $T \rightarrow F$. The RHS of this grammar production is F, which is one grammar symbol long, so we remove exactly one pair from the top of the stack (the pair that precisely contains F). Then we add a new pair at the top of the stack for the LHS of the grammar production, which is T. To compute the new state that is associated with this T in the new pair at the top of the stack, we use the GOTO part of the parsing table again: we take the state 6 that is directly to the left of T on the stack,

take the T itself, and use these two pieces of information to find the new state GOTO$[6, T]$ that should go at the top of the stack in the pair for T. In this case the new state is 9:

Stack	Input	Output
$0\ \underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{T\ 9}$	$\texttt{*}\ \texttt{id}\ \$$	reduce $T \rightarrow F$

The state at the top of the stack is now 9 and the next input symbol is `*`. The table entry ACTION$[9, *]$ is "shift 7" so we move the token `*` from the input to a new pair on the stack. The new state associated with `*` in the new pair is 7, as indicated by the shift action:

Stack	Input	Output
$0\ \underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{T\ 9}\ \underbrace{\texttt{*}\ 7}$	$\texttt{id}\ \$$	shift 7

Note that in this last step we did not reduce the T that was at the top of the stack into an E, as we did in the fourth step above. Instead we shifted the next input token `*` onto the stack. The difference is that the T here is associated with state 9, while before the T was associated with state 2. And the T is associated with state 9 here because the T has state 6 on its left, while before the T had state 0 on its left. The other difference is that here the next input token was `*`, while before the next input token was `+`. So the decision to reduce the top of the stack does not depend just on what is at the top of the stack, it depends in essence on the whole content of the stack (as summarized in essence by the state number at the top of the stack) as well as on the next input token.

The state at the top of the stack is now 7 and the next input symbol is `id`. The table entry ACTION$[7, \texttt{id}]$ is "shift 5" so we move the token `id` from the input to a new pair on the stack. The new state associated with `id` in the new pair is 5, as indicated by the shift action:

Stack	Input	Output
$0\ \underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{T\ 9}\ \underbrace{\texttt{*}\ 7}\ \underbrace{\texttt{id}\ 5}$	$\$$	shift 5

The state at the top of the stack is now 5 and the next input symbol is \$. The table entry ACTION$[5, \$]$ is "reduce 6", so we reduce the content of the stack using grammar production number 6, which is $F \rightarrow \texttt{id}$. The RHS of this grammar production is `id`, which is one grammar symbol long, so we remove exactly one pair from the top of the stack (the pair that precisely contains `id`). Then we add a new pair at the top of the stack for the LHS of the grammar production, which is F. To compute the new state that is associated with this F in the new pair at the top of the stack, we use the GOTO part of the parsing table again: we take the state 7 that is directly to the left of F on the stack, take the F itself, and use these two pieces of information to find the new state GOTO$[7, F]$ that should go at the top of the stack in the pair for F. In this case the new state is 10:

Stack	Input	Output
$0\ \underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{T\ 9}\ \underbrace{\texttt{*}\ 7}\ \underbrace{F\ 10}$	$\$$	reduce $F \rightarrow \texttt{id}$

The state at the top of the stack is now 10 and the next input symbol is again \$ (in fact it will remain \$ until the end, since all the input tokens have now been consumed). The table entry ACTION[10, \$] is "reduce 3", so we reduce the content of the stack using grammar production number 3, which is $T \rightarrow T * F$. The RHS of this grammar production is $T * F$, which is three grammar symbols long (one token and two nonterminals), so this time we remove three pairs from the top of the stack in one step (the three pairs that precisely contain $T*F$). Then we add a new pair at the top of the stack for the LHS of the grammar production, which is T. To compute the new state that is associated with this T in the new pair at the top of the stack, we use the GOTO part of the parsing table again: we take the state 6 that is directly to the left of T on the stack, take the T itself, and use these two pieces of information to find the new state GOTO[6, T] that should go at the top of the stack in the pair for T. In this case the new state is 9:

Stack	Input	Output
$0 \underbrace{E\ 1}\ \underbrace{+\ 6}\ \underbrace{T\ 9}$	\$	reduce $T \rightarrow T * F$

Note that in this last step we did not reduce the F that was at the top of the stack into a T, as we did twice before. Instead we reduced the three grammar symbols at the top of the stack all together in one step to get a T. Again, the difference in behavior is due to different state numbers associated with F (10 in the previous step, instead of 3 as before) and to different symbols appearing next in the input (\$ here, instead of + or * as before).

The state at the top of the stack is now 9 and the next input symbol is \$. The table entry ACTION[9, \$] is "reduce 1", so we reduce the content of the stack using grammar production number 1, which is $E \rightarrow E + T$. The RHS of this grammar production is $E + T$, which is three grammar symbols long (again one token and two nonterminals), so again we remove three pairs from the top of the stack in one step (the three pairs that precisely contain $E+T$). Then we add a new pair at the top of the stack for the LHS of the grammar production, which is E. To compute the new state that is associated with this E in the new pair at the top of the stack, we use the GOTO part of the parsing table again: we take the state 0 that is directly to the left of E on the stack, take the E itself, and use these two pieces of information to find the new state GOTO[0, E] that should go at the top of the stack in the pair for E. In this case the new state is 1:

Stack	Input	Output
$0 \underbrace{E\ 1}$	\$	reduce $E \rightarrow E + T$

The state at the top of the stack is now 1 and the next input symbol is still \$. The table entry ACTION[1, \$] is "accept", so the parsing process is over and the parser accepts the input sentence as a syntactically correct arithmetic expression:

Stack	Input	Output
$0 \underbrace{E\ 1}$	\$	accept

Note that at this point the input is empty (all the input tokens have been consumed) and the start symbol E of the grammar is the only grammar symbol on the stack. State 0 is at the bottom of the stack, as always, and state 1 is at the top of the stack (but this has no general meaning, state 1 is at the top of the stack simply because it is the row of the parsing table that contains the "accept" entry).

Here the complete result of the parsing algorithm:

Stack	Input	Output
0	`id + id * id` $	
0 $\underbrace{\texttt{id}\ 5}$	`+ id * id` $	shift 5
0 $\underbrace{F\ 3}$	`+ id * id` $	reduce $F \to$ `id`
0 $\underbrace{T\ 2}$	`+ id * id` $	reduce $T \to F$
0 $\underbrace{E\ 1}$	`+ id * id` $	reduce $E \to T$
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}$	`id * id` $	shift 6
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{\texttt{id}\ 5}$	`* id` $	shift 5
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{F\ 3}$	`* id` $	reduce $F \to$ `id`
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{T\ 9}$	`* id` $	reduce $T \to F$
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{T\ 9}\ \underbrace{\texttt{*}\ 7}$	`id` $	shift 7
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{T\ 9}\ \underbrace{\texttt{*}\ 7}\ \underbrace{\texttt{id}\ 5}$	$	shift 5
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{T\ 9}\ \underbrace{\texttt{*}\ 7}\ \underbrace{F\ 10}$	$	reduce $F \to$ `id`
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{T\ 9}$	$	reduce $T \to T$ `*` F
0 $\underbrace{E\ 1}$	$	reduce $E \to E$ `+` T
0 $\underbrace{E\ 1}$	$	accept

Here is then the list of grammar productions used by the parser in all the reduction steps of the parsing process just above:

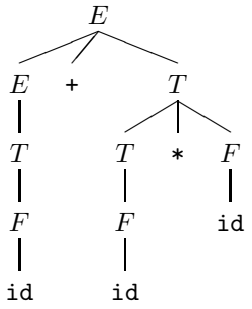
$$
\begin{aligned}
F &\to \texttt{id} \\
T &\to F \\
E &\to T \\
F &\to \texttt{id} \\
T &\to F \\
F &\to \texttt{id} \\
T &\to T\ \texttt{*}\ F \\
E &\to E\ \texttt{+}\ T
\end{aligned}
$$

If we now read this list *backwards* (from the last line back to the first one) we get the list of grammar productions to use in a rightmost derivation of the input sentence:

	E	start symbol
$\Rightarrow$	E `+` T	using $E \rightarrow E$ `+` T
$\Rightarrow$	E `+` T `*` F	using $T \rightarrow T$ `*` F
$\Rightarrow$	E `+` T `*` `id`	using $F \rightarrow$ `id`
$\Rightarrow$	E `+` F `*` `id`	using $T \rightarrow F$
$\Rightarrow$	E `+` `id` `*` `id`	using $F \rightarrow$ `id`
$\Rightarrow$	T `+` `id` `*` `id`	using $E \rightarrow T$
$\Rightarrow$	F `+` `id` `*` `id`	using $T \rightarrow F$
$\Rightarrow$	`id` `+` `id` `*` `id`	using $F \rightarrow$ `id`

Note how, while parsing some input as we have done above, the grammar symbols in the stack plus the remaining tokens in the input always form together one of the sentential forms in the rightmost derivation we have just computed. This is because the stack always represents (an abstract version of) the input that we have already processed (i.e. the past), and the tokens remaining in the input represent what the input we have still to process (i.e. the future), so together the stack and the current input represent (an abstract version of) the whole input.

Based on this rightmost derivation the following parse tree can then be built:



Implementing LR Parsers

Given the parsing algorithm above, it is easy to implement the code of an LR parser: use a one-dimensional array to implement the stack, use two two-dimensional arrays to implement the parsing table, one array for the ACTION part and one array for the GOTO part, then write C code that directly follows the algorithm by looking at the state currently at the top of the stack, what is next in the input, and taking the appropriate action. The C code itself is left as an exercise to the reader, since it is a straightforward implementation of the algorithm. The only thing different that the C code might want to do is to, instead of outputting grammar productions, directly build the corresponding parse tree by creating new parse tree nodes based on the grammar productions used.

The hard part in implementing an LR parser is therefore not writing the actual code, but rather first computing the parsing table, based on the structure of the grammar that we want the parser to use. This is what we explain in the next section.

Creating the Parsing Table for LR Parsers

Starting from a context-free grammar, there are different ways to construct a parsing table, depending on the type of LR parser we want to get as a result: LR(0), SLR(1), LALR(1), LR(1), LR(2), LR(3), etc. Here we have listed the different kinds of parsers by increasing power: each kind of parser in the list can do everything the kind of parsers on its left can do, plus more.

In practice the parsing tables for LR(k) parsers with $k \geq 2$ are not used, because the parsing tables are enormous. For example, for $k = 2$, the ACTION part of a parsing table has to have one column for each possible combination of two input symbols. For our simple grammar for arithmetic expressions that we have used above, which uses six different input symbols (five tokens plus \$), the resulting parsing table would have to have 36 columns in its ACTION part... At the opposite end of the list, LR(0) parsers are so simple that for many grammars the corresponding parsing tables contain many shift-reduce conflicts.

In this section, therefore, we are going to look mostly at how to construct SLR(1) parsing tables (SLR, for short). SLR stands for "Simple LR", and it will already be complicated enough for us. Then we will quickly discuss LR(0), LR(1), and LALR(1).

To construct an SLR parsing table from a context-free grammar, we are going to do three things: first we are going to augment the grammar, then construct a DFA from the augmented grammar based on the computation of sets of items, then we will construct the parsing table based on the transitions of that DFA. The algorithm to construct the DFA is going to be very similar to the Subset Construction algorithm we have seen for transforming an NFA into a DFA, except that it will be based on sets of items instead of sets of NFA states. But before we look at this, we first need to explain how to augment a grammar.

Augmented Context-Free Grammar

Given a grammar with a start symbol S:

$$\begin{array}{lcl} S & \rightarrow & \ldots \\ \ldots & \rightarrow & \ldots \end{array}$$

we construct the corresponding *augmented grammar* by creating a new nonterminal S' and adding a new grammar production to transform an S' into an S:

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & \ldots \\ \ldots & \rightarrow & \ldots \end{array}$$

The new nonterminal S' then becomes the new start symbol of the augmented grammar.

The new start symbol S' and the associated production $S' \rightarrow S$ seem fairly useless, but we will need them to decide when the parser should accept its input:

the parser will accept the input exactly when it is about to reduce S on its stack into an S' using the new grammar production above.

LR(0) Items

To construct the parsing table for an SLR(1) parser, we need to define LR(0) items. As the name indicates, these LR(0) items are used in the construction of LR(0) parsers, but it turns out that SLR(1) parsers use the exact same items, hence why we need them here.

An *LR(0) item* (item, for short) is simply a grammar production with a dot somewhere in its RHS. For example, from the grammar production $A \to XYZ$ we can create four different items:

$$\begin{array}{lcl} A & \to & \bullet XYZ \\ A & \to & X\bullet YZ \\ A & \to & XY\bullet Z \\ A & \to & XYZ\bullet \end{array}$$

An item like $A \to X\bullet YZ$ simply means that the parser is currently in the middle of matching an A in the input, that it has already matched the X that forms the first part of an A (that X will therefore be on the stack), and that it still has to match YZ with what remains in the input.

As a special case, the grammar production $A \to \varepsilon$ only generates one single item: $A \to \bullet$.

Later we are going to use sets of such items to compute the different possible states of the parser. For example, if the parser is in a state containing the item $A \to X\bullet YZ$ and the parser matches Y next in the input, the parser will then move into a state containing the item $A \to XY\bullet Z$.

Closure of a Set of Items

If a parser is in a state corresponding to the item $A \to X\bullet YZ$ then the parser has already matched X in the input and expects to match Y next. But the grammar might have a production like $Y \to UVW$, so, in order to match a Y, the parser might first have to match UVW, then reduce UVW into Y. This means that, if the parser expects to match Y next in the input, it should also be ready to match U next in the input, since the Y can start with U. So the parser state corresponding to the item $A \to X\bullet YZ$ should also correspond to the item $Y \to \bullet UVW$.

To take this problem into account, we compute the *closure* of a set of items I using the following algorithm.

- Every item in I is in closure(I).
- If $A \to \alpha\bullet B\beta$ is an item in closure(I) and if $B \to \gamma$ is a grammar production, where A and B are nonterminals and α, β, and γ are any sequence of terminals and nonterminals, then add the item $B \to \bullet\gamma$ to closure(I).
- Repeat the previous step until closure(I) does not change anymore.

Note how similar this is to computing the closure of an NFA state in the Subset Construction algorithm, except that here we are dealing with items and grammar productions instead of dealing with NFA states and ε-transitions.

For example, let's consider the augmented version of our grammar for simple arithmetic expressions:

$$\begin{array}{lcl} E' & \rightarrow & E \\ E & \rightarrow & E \texttt{ + } T \\ E & \rightarrow & T \\ T & \rightarrow & T \texttt{ * } F \\ T & \rightarrow & F \\ F & \rightarrow & \texttt{(} E \texttt{)} \\ F & \rightarrow & \texttt{id} \end{array}$$

If we define I as being the set of items $\{E' \rightarrow \bullet E\}$ then we compute closure(I) as follows. First, since $E' \rightarrow \bullet E$ is in I, then this item is also in closure(I). Since this first item in closure(I) has a dot directly on the left of the nonterminal E, we have then to look at the grammar productions for this nonterminal E. The grammar production $E \rightarrow E\texttt{+}T$ gives us the item $E \rightarrow \bullet E\texttt{+}T$, which we add to closure(I). The grammar production $E \rightarrow T$ gives us the item $E \rightarrow \bullet T$, which we add to closure(I) as well.

The item $E \rightarrow \bullet E\texttt{+}T$ which is now in closure(I) has a dot directly on the left of the nonterminal E, so we have then to look at the grammar productions for this nonterminal E. In fact that is what we just did at the previous step, so there is nothing else to do for that item $E \rightarrow \bullet E\texttt{+}T$. The item $E \rightarrow \bullet T$ which is now in closure(I) has a dot directly on the left of the nonterminal T, so we have then to look at the grammar productions for this nonterminal T. The grammar production $T \rightarrow T\texttt{*}F$ gives us the item $T \rightarrow \bullet T\texttt{*}F$, which we add to closure(I). The grammar production $T \rightarrow F$ gives us the item $T \rightarrow \bullet F$, which we add to closure(I) as well.

The item $T \rightarrow \bullet T\texttt{*}F$ which is now in closure(I) has a dot directly on the left of the nonterminal T, so we have then to look at the grammar productions for this nonterminal T. In fact that is what we just did at the previous step, so there is nothing else to do for that item $T \rightarrow \bullet T\texttt{*}F$. The item $T \rightarrow \bullet F$ which is now in closure(I) has a dot directly on the left of the nonterminal F, so we have then to look at the grammar productions for this nonterminal F. The grammar production $F \rightarrow \texttt{(}E\texttt{)}$ gives us the item $F \rightarrow \bullet\texttt{(}E\texttt{)}$, which we add to closure(I). The grammar production $F \rightarrow \texttt{id}$ gives us the item $F \rightarrow \bullet\texttt{id}$, which we add to closure(I) as well.

The last two items we added to closure(I) have each a dot directly on the left of a token, not on the left of a nonterminal, so at this point the computation of the closure stops, and the closure of I is therefore closure(I) = $\{E' \rightarrow \bullet E,\ E \rightarrow \bullet E\texttt{+}T,\ E \rightarrow \bullet T,\ T \rightarrow \bullet T\texttt{*}F,\ T \rightarrow \bullet F,\ F \rightarrow \bullet\texttt{(}E\texttt{)},\ F \rightarrow \bullet\texttt{id}\}$.

This means that, if, while parsing an E', a parser is trying to match an E next in the input, the parser might in fact have to match first a T, or an F, or an `(`, or an `id`.

Goto Function for Sets of Items

In order to help us compute the SLR parsing table for a context-free (augmented) grammar, we define a *goto* function as follows.

For all the items of the form $A \to \alpha \bullet X \beta$ in a set of items I, where X is any grammar symbol (token or nonterminal), we define goto(I, X) as the *closure* of the set of items of the form $A \to \alpha X \bullet \beta$.

In other words, if the parser is currently in a state corresponding to the item $A \to \alpha \bullet X \beta$ and the parser next matches an X in the input, then the parser moves to a new state corresponding to the item $A \to \alpha X \bullet \beta$. Since β can start with many different nonterminals, we then take the closure of $A \to \alpha X \bullet \beta$ to take all the different possibilities into account.

Note how similar this is to computing the closure of a set of NFA states after doing a transition, in the Subset Construction algorithm: the NFA is in one state, does a transition on X, and we then compute the closure of the new state to have the list of all the possible states in which the NFA can be after the transition. The difference is that here we move from set of items to set of items on matching X instead of moving from set of NFA states to set of NFA states on a transition on X, and we use the closure of a set of items instead of using an ε-closure. In essence, computing the goto function is similar to computing the transition function $move_D$ of the DFA.

For example, given the set of items $I = \{E' \to E\bullet,\ E \to E\bullet\texttt{+}T\}$, we compute goto$(I, \texttt{+})$ as follows. First, for each item in the set I, we check whether it has a dot directly on the left of a `+` (which is our X in this example). The item $E' \to E\bullet$ does not have its dot directly on the left of a `+` so we can forget about it. The item $E \to E\bullet\texttt{+}T$ does have a dot directly on the left of `+`, so we create a new item by moving the dot from the left of the `+` to the right of the `+` and we then put this new item $E \to E\texttt{+}\bullet T$ into a new set of items I_{new}. We have then considered all the items in I, so to compute the final result goto$(I, \texttt{+})$ we then simply have to compute the closure of I_{new}. Since I_{new} contains the item $E \to E\texttt{+}\bullet T$ and since this item has a dot directly on the left of T, we have to add new items for T to the set, and then in turn add new items for F (as we have seen above in the example for the computation of the closure of a set of items). We then get the result goto$(I, \texttt{+})$ = closure(I_{new}) = closure$(\{E \to E\texttt{+}\bullet T\})$ = $\{E \to E\texttt{+}\bullet T,\ T \to \bullet T\texttt{*}F,\ T \to \bullet F,\ F \to \bullet\texttt{(}E\texttt{)},\ F \to \bullet\texttt{id}\}$.

The Sets-of-Items Construction Algorithm

Now that we know how to augment a grammar and compute the closure of sets of items, we are ready to construct a DFA that simulates the transitions from state to state of an LR parser. As you can guess from the previous sections, this algorithm is going to look almost exactly like the Subset Construction algorithm we have seen before to transform an NFA into a DFA. Later we will use such DFA to construct a parsing table, and we will see that in essence the job of the DFA is to match the current content of the parser's stack and thus to decide when shifts and reductions have to occur.

The method we are going to use to construct the DFA is called the *sets-of-items construction algorithm*. It takes as input an augmented grammar and computes as output a DFA which has sets of items as states and transitions on grammar symbols. The algorithm has two main steps and relies on DFA states being either unmarked (yet to be processed by the algorithm) or marked (already processed by the algorithm). Each DFA state I_i that the algorithm computes is a *set* of LR(0) items. The algorithm is then as follows:

1. Compute $I_0 = \text{closure}(\{S' \to \bullet S\})$ and add the unmarked I_0 to S_D, where S' is the start symbol of the augmented grammar and S_D is the set of DFA states.

2. While there is an unmarked DFA state $I_i \in S_D$, do the following:

   (a) For each grammar symbol X, do the following:

      i. Compute the DFA state $I_j = \text{goto}(I_i, X)$.

      ii. If $I_j \neq \emptyset$ and $I_j \notin S_D$ then add the unmarked I_j to S_D.

      iii. If $I_j \neq \emptyset$ then define $move_D(I_i, X) = I_j$.

   (b) Mark I_i in S_D.

The first step starts the algorithm by computing the start state of the new DFA. The start state I_0 of the DFA is simply the closure of the set of items that only contains the item $S' \to \bullet S$, where S' is the start symbol of the augmented grammar. This item means that the parser is expecting to match a whole input program, since the dot is directly on the left of the start symbol S of the original (non-augmented) grammar. We have to take the closure of this set of one item to take into account the fact that an S might have many different shapes. Once we have computed I_0 we put it unmarked in the set S_D of DFA states (S_D is therefore a set of sets of items).

Since the start state I_0 of the DFA is initially unmarked, it is then processed (and therefore marked, see step 2-b in the algorithm) in the second step. In fact every DFA state is unmarked when it is created and added to the set S_D of DFA states (see step 2-a-ii) so every DFA state will at some point be processed (once and only once) by the second step of the algorithm and then marked (step 2-b).

The way the second step in the algorithm processes an unmarked DFA state I_i is by considering transitions from that state I_i on every possible grammar symbol X (step 2-a). Note that this is similar to what we were doing in the Subset Construction algorithm, where we processed the current state for every possible input symbol. Here we have to process state I_i for every symbol in the grammar, including nonterminals.

To do this (in step 2-a-i) the algorithm takes I_i, which is a set of items representing the current possible state of the parser, and uses the goto function to compute the set of items which can represent the next state of the parser after it has matched the grammar symbol X. The result is then a set of items which is named I_j. If that set of items does not already exist then it is added

to the set S_D of DFA states (step 2-a-ii). Since it is added unmarked, we know it will be processed in step 2 of the algorithm some time later. From step 2-a-i we also know there should be in the DFA a transition on X from DFA state I_i to DFA state I_j (step 2-a-iii, which corresponds to adding an edge labeled with X going from node I_i to node I_j in the diagram representing the DFA).

As step 2 of the algorithm processes unmarked states and marks them, new unmarked DFA states might be created, which in turn are processed by step 2 of the algorithm, and so on like this until no more DFA states are created and all the existing DFA states have been processed.

Note that, unlike what happened in the Subset Construction algorithm, there is no third step to compute the final states of the DFA. This is because the parser will not use the DFA's final states to decide when to accept the input. Rather the parser will rely on the presence of the extra nonterminal S' of the augmented grammar to decide when to accept the input. So there is no need here to compute final states for the DFA.

We can now look at an example to see how the sets-of-items construction algorithm actually works in practice. For this example we will of course use our usual grammar for simple arithmetic expressions, augmented with a new nonterminal E' which is the new start symbol of the grammar:

$$\begin{array}{lcl} E' & \rightarrow & E \\ E & \rightarrow & E \texttt{ + } T \\ E & \rightarrow & T \\ T & \rightarrow & T \texttt{ * } F \\ T & \rightarrow & F \\ F & \rightarrow & \texttt{(} E \texttt{)} \\ F & \rightarrow & \texttt{id} \end{array}$$

To start the algorithm we first have to compute the initial set of items I_0. To compute this set, we start with the grammar production for the start symbol in the augmented grammar $E' \rightarrow E$ and transform this grammar production into an item by placing a dot at the left end of the right hand side of the production: $E' \rightarrow \bullet E$. The set I_0 is then the closure of the set containing just this item: I_0 = closure($\{E' \rightarrow \bullet E\}$) =
$\{E' \rightarrow \bullet E,\ E \rightarrow \bullet E\texttt{+}T,\ E \rightarrow \bullet T,\ T \rightarrow \bullet T\texttt{*}F,\ T \rightarrow \bullet F,\ F \rightarrow \bullet\texttt{(}E\texttt{)},\ F \rightarrow \bullet\texttt{id}\}$.

This gives us the following DFA:

Note that, unlike what we did for the Subset Construction algorithm, in the sets-of-items construction algorithm we do not compute a table of closures in advance. That's because the augmented grammar above corresponds to 20 different items, so the table would be big, and most of the closures are trivial to compute since they are for items where the dot is directly on the left of a token (i.e. items for which the closure only contains the item itself). In practice we will only really need to compute a closure when the dot in an item will be directly on the left of an E, a T, or an F, in which cases most of the closures

will look very similar. So in this algorithm it is easier to compute closures as we need them.

Now that we have computed I_0, we have to consider each grammar symbol in turn and compute the transitions that can happen from I_0 on these grammar symbols. To compute transitions like this, we are always going to consider the grammar symbols in the same order: E, T, F, `+`, `*`, `(`, `)`, and `id`. Note that the special symbol \$ is not included in the list, since it is not part of the grammar. This special symbol will only appear later, when we actually construct the parsing table.

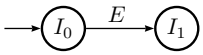
In theory we also would have to consider the grammar symbol E', since E' appears in the augmented grammar which we are using to compute the sets of items. In practice though, since we introduced E' ourselves in the grammar when augmenting it, we know that E' only appears on the left hand side of the first grammar production. A transition from some set of items I on symbol E' would only be possible if E' appeared on the right hand side of a grammar production, which is never the case. Therefore $\text{goto}(I, E')$ will always be the empty set for all I, and computing it is therefore useless.

So let's start the second step of the algorithm. We have to first compute $\text{goto}(I_0, E)$. To compute this set we have to start with the set of items I_0. For each item in this set, we compute the item which is the result of the transition on E (if any). Then we take the union of the resulting items, and compute the closure of the union:

E' → •E	$\xrightarrow{E}$	E' → E•
E → •E+T	$\xrightarrow{E}$	E → E•+T
E → •T	$\xrightarrow{E}$	$\emptyset$
T → •T*F	$\xrightarrow{E}$	$\emptyset$
T → •F	$\xrightarrow{E}$	$\emptyset$
F → •(E)	$\xrightarrow{E}$	$\emptyset$
F → •id	$\xrightarrow{E}$	$\emptyset$
union	=	$\{E' \to E\bullet,\ E \to E\bullet+T\}$
$\text{goto}(I_0, E) = \text{closure(union)}$	=	$\{E' \to E\bullet,\ E \to E\bullet+T\} = I_1$

The first two lines of the table mean that, if the parser is in a state corresponding to the item $E' \to \bullet E$ then, after matching an E, the parser will be in a state corresponding to the item $E' \to E\bullet$, and if the parser is in a state corresponding to the item $E \to \bullet E+T$ then, after matching an E, the parser will be in a state corresponding to the item $E \to E\bullet+T$. The union then means that, if the parser is in a state corresponding to any item $E' \to \bullet E$, $E \to \bullet E+T$, $E \to \bullet T$, $T \to \bullet T*F$, $T \to \bullet F$, $F \to \bullet(E)$, or $F \to \bullet\texttt{id}$ in the set I_0 then, after matching an E, the parser will be in a state corresponding to the items $E' \to E\bullet$ and $E \to E\bullet+T$. For each of these two items the dot is directly on the left of a token, so the closure of the union does not add any new item.

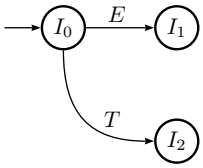
Given the table above, we create a new state for I_1 is the DFA and add a transition on E between the two states:



Next we compute goto(I_0, T):

E' $\rightarrow \bullet$E	$\xrightarrow{T}$	$\emptyset$
E $\rightarrow \bullet$E+T	$\xrightarrow{T}$	$\emptyset$
E $\rightarrow \bullet$T	$\xrightarrow{T}$	E $\rightarrow$ T$\bullet$
T $\rightarrow \bullet$T*F	$\xrightarrow{T}$	T $\rightarrow$ T$\bullet$*F
T $\rightarrow \bullet$F	$\xrightarrow{T}$	$\emptyset$
F $\rightarrow \bullet$(E)	$\xrightarrow{T}$	$\emptyset$
F $\rightarrow \bullet$id	$\xrightarrow{T}$	$\emptyset$
union	=	$\{E \rightarrow T\bullet,\ T \rightarrow T\bullet *F\}$
goto(I_0, T) = closure(union)	=	$\{E \rightarrow T\bullet,\ T \rightarrow T\bullet *F\} = I_2$

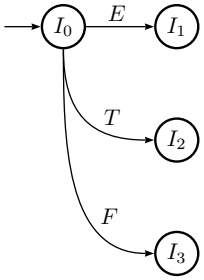
So again we add a new state and a transition on T:



Next we compute goto(I_0, F):

E' $\rightarrow \bullet$E	$\xrightarrow{F}$	$\emptyset$
E $\rightarrow \bullet$E+T	$\xrightarrow{F}$	$\emptyset$
E $\rightarrow \bullet$T	$\xrightarrow{F}$	$\emptyset$
T $\rightarrow \bullet$T*F	$\xrightarrow{F}$	$\emptyset$
T $\rightarrow \bullet$F	$\xrightarrow{F}$	T $\rightarrow$ F$\bullet$
F $\rightarrow \bullet$(E)	$\xrightarrow{F}$	$\emptyset$
F $\rightarrow \bullet$id	$\xrightarrow{F}$	$\emptyset$
union	=	$\{T \rightarrow F\bullet\}$
goto(I_0, F) = closure(union)	=	$\{T \rightarrow F\bullet\} = I_3$

So once again we add a new state and a new transition on F:



Next we have to do the same thing for all the tokens, starting with `+`:

E' → •E	$\xrightarrow{+} \emptyset$
E → •E+T	$\xrightarrow{+} \emptyset$
E → •T	$\xrightarrow{+} \emptyset$
T → •T*F	$\xrightarrow{+} \emptyset$
T → •F	$\xrightarrow{+} \emptyset$
F → •(E)	$\xrightarrow{+} \emptyset$
F → •id	$\xrightarrow{+} \emptyset$
union	$= \emptyset$
goto(I_0, +) = closure(union)	$= \emptyset$

Here the set of items we just computed is empty. This means that, if the parser expects an E, or a T, or an F, or an `(`, or an `id` next in the input (as indicated by what is directly on the right of the dot in each item in I_0), and if the next input token is `+`, then the parser cannot do anything: a syntax error is then detected.

So we leave the DFA unchanged and then compute goto(I_0, *):

E' → •E	$\xrightarrow{*} \emptyset$
E → •E+T	$\xrightarrow{*} \emptyset$
E → •T	$\xrightarrow{*} \emptyset$
T → •T*F	$\xrightarrow{*} \emptyset$
T → •F	$\xrightarrow{*} \emptyset$
F → •(E)	$\xrightarrow{*} \emptyset$
F → •id	$\xrightarrow{*} \emptyset$
union	$= \emptyset$
goto(I_0, *) = closure(union)	$= \emptyset$

and again there is nothing to do.

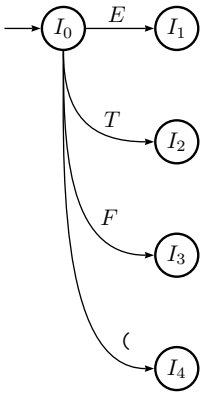
Next we compute goto(I_0, ():

E' → •E	$\xrightarrow{(}$	$\emptyset$
E → •E+T	$\xrightarrow{(}$	$\emptyset$
E → •T	$\xrightarrow{(}$	$\emptyset$
T → •T*F	$\xrightarrow{(}$	$\emptyset$
T → •F	$\xrightarrow{(}$	$\emptyset$
F → •(E)	$\xrightarrow{(}$	F → (•E)
F → •id	$\xrightarrow{(}$	$\emptyset$
union	=	$\{F \to (\bullet E)\}$
goto(I_0, () = closure(union)	=	I_4

where I_4 is $\{F \to (\bullet E),\ E \to \bullet E+T,\ E \to \bullet T,\ T \to \bullet T*F,\ T \to \bullet F,\ F \to \bullet(E),\ F \to \bullet\texttt{id}\}$.

Here the union contains the item $F \to (\bullet E)$, which has a dot directly on the left of E. This means that computing the closure for this item introduces many new items corresponding to the grammar productions for E, then T, then F.

Based on the previous table we then create a new state of the DFA and add a new transition on (:



Then we compute goto(I_0,)):

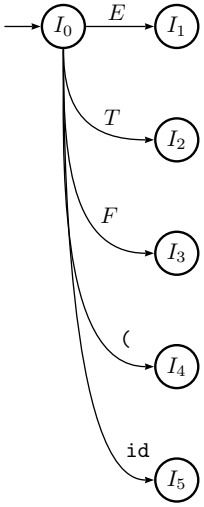
E' → •E	$\xrightarrow{)}$	$\emptyset$
E → •E+T	$\xrightarrow{)}$	$\emptyset$
E → •T	$\xrightarrow{)}$	$\emptyset$
T → •T*F	$\xrightarrow{)}$	$\emptyset$
T → •F	$\xrightarrow{)}$	$\emptyset$
F → •(E)	$\xrightarrow{)}$	$\emptyset$
F → •id	$\xrightarrow{)}$	$\emptyset$
union	=	$\emptyset$
goto(I_0,)) = closure(union)	=	$\emptyset$

and there is nothing to do.

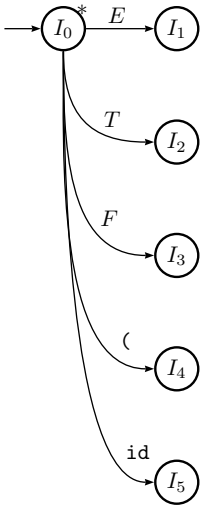
Finally we compute goto(I_0, `id`):

E' $\rightarrow \bullet$E	$\overset{\texttt{id}}{\rightarrow}$	$\emptyset$
E $\rightarrow \bullet$E+T	$\overset{\texttt{id}}{\rightarrow}$	$\emptyset$
E $\rightarrow \bullet$T	$\overset{\texttt{id}}{\rightarrow}$	$\emptyset$
T $\rightarrow \bullet$T*F	$\overset{\texttt{id}}{\rightarrow}$	$\emptyset$
T $\rightarrow \bullet$F	$\overset{\texttt{id}}{\rightarrow}$	$\emptyset$
F $\rightarrow \bullet$(E)	$\overset{\texttt{id}}{\rightarrow}$	$\emptyset$
F $\rightarrow \bullet$`id`	$\overset{\texttt{id}}{\rightarrow}$	F $\rightarrow$ `id`$\bullet$
union	=	$\{F \rightarrow \texttt{id}\bullet\}$
goto(I_0, `id`) = closure(union)	=	$\{F \rightarrow \texttt{id}\bullet\} = I_5$

so we add a new state again and a new transition on `id`:



At this point in the algorithm we have processed I_0 for all the grammar symbols, so we can mark I_0 as processed:



Now we just have to repeat the same process for all the other states that remain to be processed, by considering them one by one for each of the eight grammar symbols E, T, F, `+`, `*`, `(`, `)`, and `id`. As you can see, the process is quite simple, but is very long and takes a lot of time, which is why in practice people use tools like YACC to do this computation automatically.

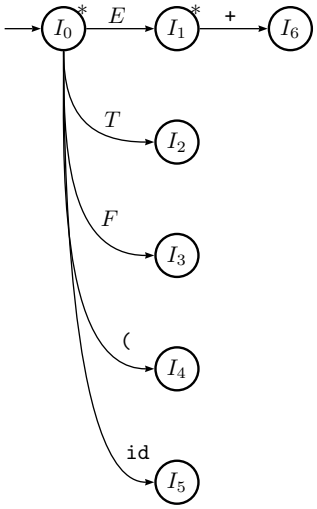
In order to speed up the computation, we can be a little clever and notice that the result of the computation $\text{goto}(I, X)$ is going to be non-empty only if there is in I an item with a dot directly on the left of an X. Since I_1 is the set of items $\{E' \to E\bullet,\ E \to E\bullet\texttt{+}T\}$ and since in this set of items the only grammar symbol with a dot directly on its left is `+`, we therefore know that $\text{goto}(I_1, X)$ is going to be non-empty only when X is `+`. For all the other grammar symbols $\text{goto}(I_0, X)$ will be empty and the DFA will not change.

So we can compute $\text{goto}(I_1, \texttt{+})$:

E' $\to$ E$\bullet$	$\overset{\texttt{+}}{\to}$	$\emptyset$
E $\to$ E$\bullet$+T	$\overset{\texttt{+}}{\to}$	E $\to$ E+$\bullet$T
union	=	$\{E \to E\texttt{+}\bullet T\}$
goto(I_1, +) = closure(union)	=	I_6

where I_6 is $\{E \to E\texttt{+}\bullet T,\ T \to \bullet T\texttt{*}F,\ T \to \bullet F,\ F \to \bullet\texttt{(}E\texttt{)},\ F \to \bullet\texttt{id}\}$.

We can then add a new state for I_6, add a new transition on `+` from I_1 to I_6, and directly mark I_1 as processed, since for all the other grammar symbols we know that the DFA will not change:

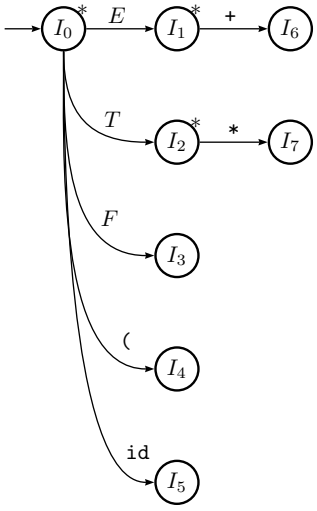


Similarly, I_2 is $\{E \to T\bullet,\ T \to T\bullet * F\}$ and in this set of items only the grammar symbol $*$ has a dot directly on its left. Therefore we only need to compute goto$(I_2, *)$ and we know that there will be nothing to do for all the other grammar symbols.

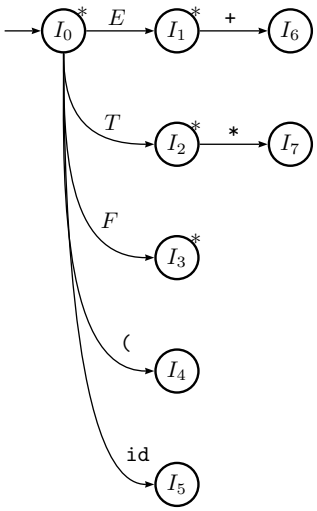
E $\to$ T$\bullet$	$\overset{*}{\to}$	$\emptyset$
T $\to$ T$\bullet$*F	$\overset{*}{\to}$	T $\to$ T*$\bullet$F
union	=	$\{T \to T*\bullet F\}$
goto$(I_2, *)$ = closure(union)	=	I_7

where I_7 is $\{T \to T*\bullet F,\ F \to \bullet(E),\ F \to \bullet\texttt{id}\}$.

So we can directly add a new state I_7, a new transition from I_2 to I_7 on $*$, and mark I_2 as processed:



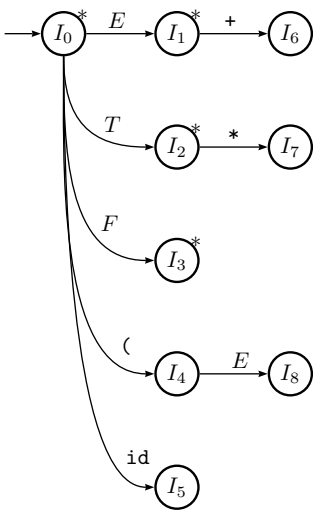
Then we have to process I_3, which is $\{T \to F\bullet\}$. This set of items does not contain any item which has a grammar symbol with a dot directly on its left, since the only item in I_3 has the dot at the right end of the RHS. This means that goto(I_3, X) is the empty set for all X, therefore there cannot be any transition out of I_3, and we can directly mark I_3 as processed:



Next we have to process I_4. First on E:

F $\to$ (•E)	$\xrightarrow{E}$	F $\to$ (E•)
E $\to$ •E+T	$\xrightarrow{E}$	E $\to$ E•+T
E $\to$ •T	$\xrightarrow{E}$	$\emptyset$
T $\to$ •T*F	$\xrightarrow{E}$	$\emptyset$
T $\to$ •F	$\xrightarrow{E}$	$\emptyset$
F $\to$ •(E)	$\xrightarrow{E}$	$\emptyset$
F $\to$ •id	$\xrightarrow{E}$	$\emptyset$
union	=	$\{F \to (E\bullet),\ E \to E\bullet+T\}$
goto(I_4, E) = closure(union)	=	$\{F \to (E\bullet),\ E \to E\bullet+T\} = I_8$

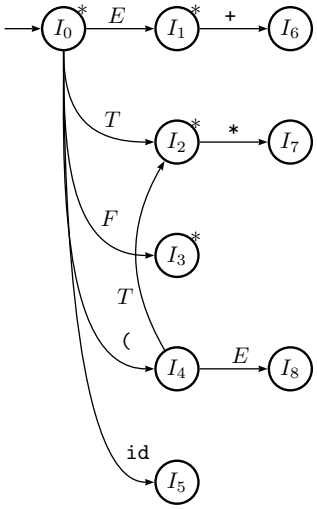
so we add a new state and a new transition:



Next we compute goto(I_4, T):

F $\rightarrow$ (•E)	$\xrightarrow{T}$	$\emptyset$
E $\rightarrow$ •E+T	$\xrightarrow{T}$	$\emptyset$
E $\rightarrow$ •T	$\xrightarrow{T}$	E $\rightarrow$ T•
T $\rightarrow$ •T*F	$\xrightarrow{T}$	T $\rightarrow$ T•*F
T $\rightarrow$ •F	$\xrightarrow{T}$	$\emptyset$
F $\rightarrow$ •(E)	$\xrightarrow{T}$	$\emptyset$
F $\rightarrow$ •id	$\xrightarrow{T}$	$\emptyset$
union	=	I_2
goto(I_4, T) = closure(union)	=	I_2

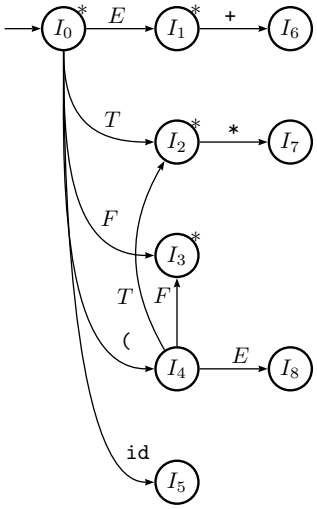
Here the union we compute is the same set of items as I_2 and therefore the closure of the union is the same set I_2 as well (since I_2 is itself the result of computing a closure when we computed goto(I_0, T)). So here there is no need to create a new state, we simply create a new transition on T from I_4 back to I_2:



We now compute goto(I_4, F):

F → (•E)	$\xrightarrow{F}$	$\emptyset$
E → •E+T	$\xrightarrow{F}$	$\emptyset$
E → •T	$\xrightarrow{F}$	$\emptyset$
T → •T*F	$\xrightarrow{F}$	$\emptyset$
T → •F	$\xrightarrow{F}$	T → F•
F → •(E)	$\xrightarrow{F}$	$\emptyset$
F → •id	$\xrightarrow{F}$	$\emptyset$
union	=	I_3
goto(I_4, F) = closure(union)	=	I_3

Again there is no need to create a new state, we just add a new transition:

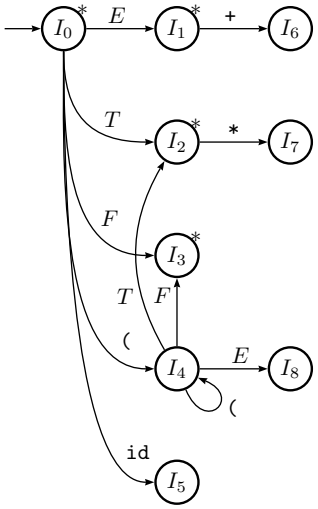


Since I_4 is $\{F \to \texttt{(}\bullet E\texttt{)},\ E \to \bullet E\texttt{+}T,\ E \to \bullet T,\ T \to \bullet T\texttt{*}F,\ T \to \bullet F,\ F \to \bullet\texttt{(}E\texttt{)},\ F \to \bullet\texttt{id}\}$, I_4 does not contain any item which has a dot directly on the left of either `+`, `*`, or `)`, so we can skip the computation of goto(I_4, X) for all these three grammar symbols.

For goto(I_4, `(`) we get the following:

F → (•E)	↷	∅
E → •E+T	↷	∅
E → •T	↷	∅
T → •T*F	↷	∅
T → •F	↷	∅
F → •(E)	↷	F → (•E)
F → •id	↷	∅
union	=	$\{F \to \texttt{(}\bullet E\texttt{)}\}$
goto(I_4, () = closure(union)	=	I_4

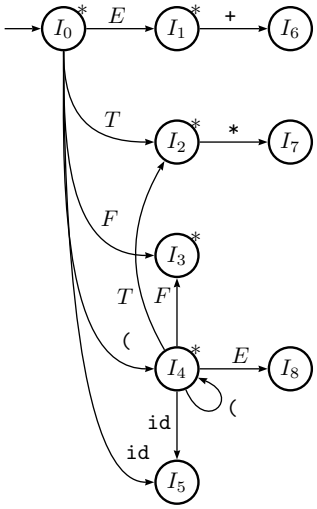
In this case the union contains just the item $F \to \texttt{(}\bullet E\texttt{)}$, but, since this item has a dot directly on the left of E, computing the closure of the union adds many new items to the set, and the result is I_4 itself. We therefore add a loop to the DFA, from I_4 to itself on a transition on `(`:



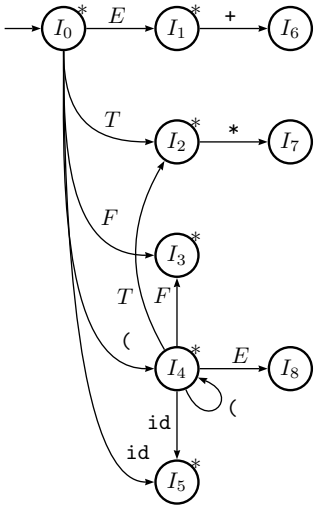
Finally for goto(I_4, `id`) we get the following:

F → (•E)	$\xrightarrow{\texttt{id}}$	$\emptyset$
E → •E+T	$\xrightarrow{\texttt{id}}$	$\emptyset$
E → •T	$\xrightarrow{\texttt{id}}$	$\emptyset$
T → •T*F	$\xrightarrow{\texttt{id}}$	$\emptyset$
T → •F	$\xrightarrow{\texttt{id}}$	$\emptyset$
F → •(E)	$\xrightarrow{\texttt{id}}$	$\emptyset$
F → •id	$\xrightarrow{\texttt{id}}$	F → id•
union	=	I_5
goto(I_4, `id`) = closure(union)	=	I_5

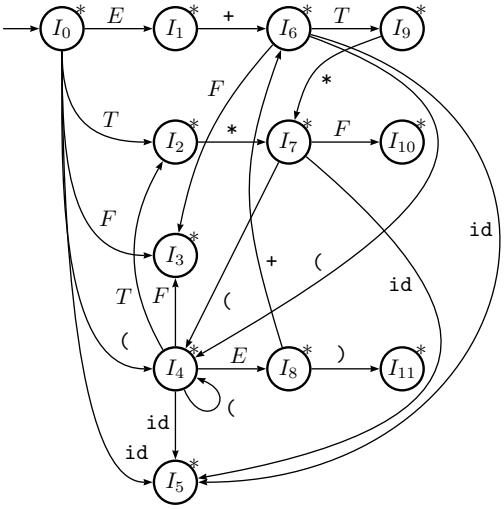
so we just add a new transitions from I_4 to I_5 on `id` and then mark I_4 as processed:



Then we have to process I_5, which is $\{F \to \texttt{id}\bullet\}$. Just like I_3, this set of items does not contain any item which has a grammar symbol with a dot directly on its left, since the only item in I_5 has the dot at the right end of the RHS. This means that goto(I_5, X) is the empty set for all X, therefore there cannot be any transition out of I_5, and we can directly mark I_5 as processed:



We keep proceeding like this, computing goto(I, X) for all unmarked I and all grammar symbols X, until all the stated in the DFA have been marked. The final version of the DFA is then:



with the following sets of items:

i	I_i
0	$\{E' \to \bullet E,\ E \to \bullet E\texttt{+}T,\ E \to \bullet T,\ T \to \bullet T\texttt{*}F,\ T \to \bullet F,\ F \to \bullet\texttt{(}E\texttt{)},\ F \to \bullet\texttt{id}\}$
1	$\{E' \to E\bullet,\ E \to E\bullet\texttt{+}T\}$
2	$\{E \to T\bullet,\ T \to T\bullet\texttt{*}F\}$
3	$\{T \to F\bullet\}$
4	$\{F \to \texttt{(}\bullet E\texttt{)},\ E \to \bullet E\texttt{+}T,\ E \to \bullet T,\ T \to \bullet T\texttt{*}F,\ T \to \bullet F,\ F \to \bullet\texttt{(}E\texttt{)},\ F \to \bullet\texttt{id}\}$
5	$\{F \to \texttt{id}\bullet\}$
6	$\{E \to E\texttt{+}\bullet T,\ T \to \bullet T\texttt{*}F,\ T \to \bullet F,\ F \to \bullet\texttt{(}E\texttt{)},\ F \to \bullet\texttt{id}\}$
7	$\{T \to T\texttt{*}\bullet F,\ F \to \bullet\texttt{(}E\texttt{)},\ F \to \bullet\texttt{id}\}$
8	$\{F \to \texttt{(}E\bullet\texttt{)},\ E \to E\bullet\texttt{+}T\}$
9	$\{E \to E\texttt{+}T\bullet,\ T \to T\bullet\texttt{*}F\}$
10	$\{T \to T\texttt{*}F\bullet\}$
11	$\{F \to \texttt{(}E\texttt{)}\bullet\}$

As we have said above when presenting the algorithm, the sets-of-items construction algorithm does not compute final states for the DFA, since we will have no use for them.

It is interesting to compare the structure of the DFA we have just computed with the way the parser uses its stack to parse input. For example, we have previously seen how the parser parses input like `id + id * id`. Here is again part of this process:

Stack	Input	Output
...	...	...
0 $\underbrace{E}$ 1 $\underbrace{\texttt{+}}$ 6 $\underbrace{T}$ 9 $\underbrace{\texttt{*}}$ 7	`id` $	...
0 $\underbrace{E}$ 1 $\underbrace{\texttt{+}}$ 6 $\underbrace{T}$ 9 $\underbrace{\texttt{*}}$ 7 $\underbrace{\texttt{id}}$ 5	$	shift 5
0 $\underbrace{E}$ 1 $\underbrace{\texttt{+}}$ 6 $\underbrace{T}$ 9 $\underbrace{\texttt{*}}$ 7 $\underbrace{F}$ 10	$	reduce $F \to$ `id`
0 $\underbrace{E}$ 1 $\underbrace{\texttt{+}}$ 6 $\underbrace{T}$ 9	$	reduce $T \to T$ `*` F

If you compare the content of the stack on the first line with the DFA above, you can see that the content of the stack describes a path in the DFA that starts with state 0, does a transition on E to state 1 of the DFA, then does a transition on `+` from state 1 to state 6, followed by a transition on T from state 6 to state 9, followed by a transition on `*` from state 9 to state 7. Shifting the token `id` then makes the DFA move from state 7 to state 5. This is why the shift action is "shift 5" in the corresponding parsing table.

When the token `id` is reduced into an F in the next step, the DFA in essence goes back a number of transitions equal to the length of the right hand side of the grammar production used, then goes forward again through the transition for the left hand side of the production. So, when the parser reduces the top of the stack using the grammar production $F \rightarrow$ `id`, the DFA goes one step backward through the transition on `id` from state 5 back to state 7, then forward again from state 7 to state 10 through the transition on F. When the parser reduces the top of the stack using the grammar production $T \rightarrow T$ `*` F, the DFA goes three steps backward through the transitions on F, `*`, and T from state 10 back to state 7 then 9 then 6, then the DFA goes forward again from state 6 to state 9 through the transition on T corresponding to the LHS of the grammar production used in the reduction.

So at every point in time, the stack describes a path in the DFA that goes from state 0 to the current state at the top of stack through a certain number of transitions in the DFA. A shift corresponds to making the current path one step longer by adding to the end of the path one more transition on a token, while a reduction makes the path shorter by replacing the end part of the path with a single transition on a nonterminal.

Note that the parser does not explicitly use the DFA, rather the structure of the DFA is implicitly encoded in the shift and reduce actions and the GOTO part of the parsing table used by the parser. We describe how this is done in the next section.

Finally, note also that, as we have said before, there is no real need in practice to keep on the stack of the parser both the grammar symbols and the state numbers: the state numbers by themselves provide enough information to the parser about the input it has processed so far. The reason for that is because, if you look at the DFA, you can decide the complete path from state 0 to the current state just by knowing the numbers of all the intermediate states. You do not need to know which grammar symbols are used for the various transitions in the path, since this information can be recovered just by looking at the sequence of states in the path. For example, if the stack says that the current state at the top of the stack is 9 and that the previous state on the stack is 6, then the grammar symbol that should be associated with the state 9 at the top of the stack obviously has to be T, since the only transition in the DFA from state 6 to state 9 is a transition on T. Keeping the grammar symbols on the stack along with the state numbers is therefore redundant. In these lecture notes we keep the grammar symbols and state numbers together in pairs on the stack to make it easier to understand what the parser does with the stack, but a real parser (generated by YACC, for example) would only keep on the stack

the state numbers, not the associated grammar symbols.

Constructing the SLR(1) Parsing Table

Now that we have computed the sets of items for the augmented grammar as well as the corresponding DFA, we can construct the SLR parsing table. The table will have one row for each set of items we computed above (one row for each state in the DFA). In the columns of the ACTION part of the table will be all the tokens and \$. In the columns of the GOTO part of the parsing table will be all the nonterminals (except for the start symbol of the augmented grammar).

We use the following three rules to construct the ACTION part of the SLR table:

- if $A \to \alpha \bullet \mathtt{a} \beta$ (where $\mathtt{a}$ is a token) is an item in the set of items I_i and $\text{goto}(I_i, \mathtt{a}) = I_j$ then set ACTION$[i, \mathtt{a}]$ to "shift j";
- if $A \to \alpha \bullet$ is an item in the set of items I_i then set ACTION$[i, \mathtt{a}]$ to "reduce $A \to \alpha$" for all $\mathtt{a}$ in FOLLOW(A) (in practice we will give a number to each grammar production $A \to \alpha$ and put in the table "reduce" followed by the production's number);
- if $S' \to S \bullet$ (where S' is the start symbol of the augmented grammar) is an item in the set of items I_i then set ACTION$[i, \$]$ to "accept";

and we use the following rule to construct the GOTO part of the SLR table:

- if $\text{goto}(I_i, A) = I_j$, where A is a nonterminal, then set GOTO$[i, A]$ to "j".

The first rule takes care of all the transitions on tokens in the DFA. These transitions directly become shift actions in the parsing table.

The second rule explains when the parser does a reduction: if an item has the dot at its right end, then the parser has just finished matching all the parts of the RHS of the corresponding grammar production. The parser can then reduce that RHS into the nonterminal A which is the LHS of the production. The reduction only occurs if the next token in the input is in FOLLOW(A) though: if the next token in the input is not in FOLLOW(A) then the parser should not reduce the top of the stack into A, otherwise the parser would end up in a configuration that corresponds to having an A followed by a token which cannot follow an A...

The third rule determines when the parser accepts the input: the parser accepts the input exactly when it has finished matching an S (the dot is directly to the right of the S in the item) where S is the start symbol of the original grammar, with no remaining token in the input (\$ is the last input symbol remaining), and is about to reduce this S into an S' where S' is the start symbol of the augmented grammar. As we have explained before, determining when to accept the input is precisely why we augmented the grammar in the first place, and this third rule is the rule that makes the determination.

Finally the fourth rule takes care of all the transitions on nonterminals in the DFA. These transitions directly become entries in the GOTO part of the

parsing table. The first and fourth rule together therefore encode the complete DFA into the parsing table.

Now that we have these rules, we can apply them to our example DFA above and construct the corresponding SLR parsing table. Before we do this though, we have two extra things to do:

- give a number to each production in the augmented grammar;
- compute the FOLLOW sets for all the nonterminals in the grammar.

Here is how we number the grammar productions in the augmented grammar:

$$
\begin{array}{llll}
0 & E' & \rightarrow & E \\
1 & E & \rightarrow & E \texttt{ + } T \\
2 & E & \rightarrow & T \\
3 & T & \rightarrow & T \texttt{ * } F \\
4 & T & \rightarrow & F \\
5 & F & \rightarrow & \texttt{(} E \texttt{)} \\
6 & F & \rightarrow & \texttt{id}
\end{array}
$$

We start with the number 0 for the first production (the one introduced by us when we augmented the grammar). From the way we are going to construct the parsing table, we know that this grammar production will never be used (it will only correspond to the "accept" entry) therefore giving it a number is a bit useless but harmless.

To compute the FOLLOW sets for the nonterminals E, T, and F, we use the exact same method as we used when studying top-down parser. Here is directly the result:

X	E	T	F
FOLLOW(X)	{+,), \$}	{+, *,), \$}	{+, *,), \$}

Note that E' is a nonterminal in our grammar too, but again E' is never used in the construction of the parsing table (except for the "accept" case) so there is no need to compute the FOLLOW set for E' (which is trivial to compute anyway, since it is always the same set as the one for E).

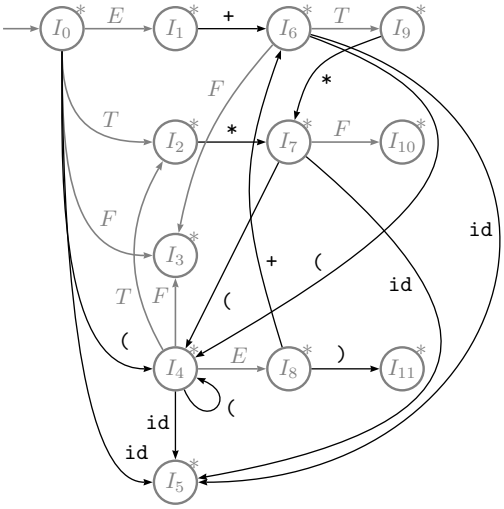
We can now build the SLR(1) parsing table. It has one row for each set of items we computed (in other words, one row for each state in the DFA). In the ACTION part the table has one column for each token plus \$, and in the GOTO part the table has one column for each nonterminal (except for E', for which there are never any transition in the DFA):

State	ACTION						GOTO		
	`id`	`+`	`*`	`(`	`)`	$	E	T	F
0									
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									

For the first rule above, we have to identify in our sets of items all the items that have a dot directly on the left of a token. Here they are, underlined:

i	I_i
0	$\{E' \to \bullet E,\ E \to \bullet E\texttt{+}T,\ E \to \bullet T,\ T \to \bullet T\texttt{*}F,\ T \to \bullet F,\ \underline{F \to \bullet\texttt{(}E\texttt{)}},\ \underline{F \to \bullet\texttt{id}}\}$
1	$\{E' \to E\bullet,\ \underline{E \to E\bullet\texttt{+}T}\}$
2	$\{E \to T\bullet,\ \underline{T \to T\bullet\texttt{*}F}\}$
3	$\{T \to F\bullet\}$
4	$\{F \to \texttt{(}\bullet E\texttt{)},\ E \to \bullet E\texttt{+}T,\ E \to \bullet T,\ T \to \bullet T\texttt{*}F,\ T \to \bullet F,\ \underline{F \to \bullet\texttt{(}E\texttt{)}},\ \underline{F \to \bullet\texttt{id}}\}$
5	$\{F \to \texttt{id}\bullet\}$
6	$\{E \to E\texttt{+}\bullet T,\ T \to \bullet T\texttt{*}F,\ T \to \bullet F,\ \underline{F \to \bullet\texttt{(}E\texttt{)}},\ \underline{F \to \bullet\texttt{id}}\}$
7	$\{T \to T\texttt{*}\bullet F,\ \underline{F \to \bullet\texttt{(}E\texttt{)}},\ \underline{F \to \bullet\texttt{id}}\}$
8	$\{\underline{F \to \texttt{(}E\bullet\texttt{)}},\ \underline{E \to E\bullet\texttt{+}T}\}$
9	$\{E \to E\texttt{+}T\bullet,\ \underline{T \to T\bullet\texttt{*}F}\}$
10	$\{T \to T\texttt{*}F\bullet\}$
11	$\{F \to \texttt{(}E\texttt{)}\bullet\}$

These thirteen items with a dot directly on the left of a token directly correspond to the thirteen transitions on a token in the DFA:



For each of these thirteen transitions, the first rule above tells us to put a corresponding shift action in the table: if there is transition on token `a` from I_i to I_j in the DFA then we put "shift j" in ACTION$[i, \texttt{a}]$.

State	ACTION						GOTO		
	`id`	`+`	`*`	`(`	`)`	$	E	T	F
0	s5			s4					
1		s6							
2			s7						
3									
4	s5			s4					
5									
6	s5			s4					
7	s5			s4					
8		s6			s11				
9			s7						
10									
11									

Note that the column for $ never contains a shift action, since, once the parser has reached the end of the input, there is nothing else the parser could possibly shift!

For the second rule above, we have to identify in our sets of items all the items that have a dot at the right end of the RHS of the item. Here they are, underlined again:

i	I_i
0	$\{E' \to \bullet E,\ E \to \bullet E\texttt{+}T,\ E \to \bullet T,\ T \to \bullet T\texttt{*}F,\ T \to \bullet F,\ F \to \bullet\texttt{(}E\texttt{)},\ F \to \bullet\texttt{id}\}$
1	$\{E' \to E\bullet,\ E \to E\bullet\texttt{+}T\}$
2	$\{\underline{E \to T\bullet},\ T \to T\bullet\texttt{*}F\}$
3	$\{\underline{T \to F\bullet}\}$
4	$\{F \to \texttt{(}\bullet E\texttt{)},\ E \to \bullet E\texttt{+}T,\ E \to \bullet T,\ T \to \bullet T\texttt{*}F,\ T \to \bullet F,\ F \to \bullet\texttt{(}E\texttt{)},\ F \to \bullet\texttt{id}\}$
5	$\{\underline{F \to \texttt{id}\bullet}\}$
6	$\{E \to E\texttt{+}\bullet T,\ T \to \bullet T\texttt{*}F,\ T \to \bullet F,\ F \to \bullet\texttt{(}E\texttt{)},\ F \to \bullet\texttt{id}\}$
7	$\{T \to T\texttt{*}\bullet F,\ F \to \bullet\texttt{(}E\texttt{)},\ F \to \bullet\texttt{id}\}$
8	$\{F \to \texttt{(}E\bullet\texttt{)},\ E \to E\bullet\texttt{+}T\}$
9	$\{\underline{E \to E\texttt{+}T\bullet},\ T \to T\bullet\texttt{*}F\}$
10	$\{\underline{T \to T\texttt{*}F\bullet}\}$
11	$\{\underline{F \to \texttt{(}E\texttt{)}\bullet}\}$

We have six such items with a dot at the right end of the RHS of the item. In fact each of these six items corresponds to one grammar production to use in the corresponding reduction. Note that we did not underline the item $E' \to E\bullet$ in I_1, even though this item has a dot at the right end of its RHS too. That's because this item is going to be used later in the third rule to determine when the parser accepts.

For each of the six items that we have underlined, we have to add "reduce" actions to the parsing table, in the rows for the corresponding sets of items and in the columns for the tokens that can follow the LHS of the items. For example, the first item we have underlined just above is $E \to T\bullet$ in I_2. This means that we have to put reduce actions for the grammar production $E \to T$ (grammar production number 2) somewhere in row number 2 of the parsing table. To decide in which columns to put the reduce actions we have to look at FOLLOW(E), which is {`+`, `)`, \$}. So we put reduce actions "r2" (i.e. "reduce using grammar production number 2") in the row number 2 of the parsing table (since the item $E \to T\bullet$ is in I_2) and in the columns for `+`, `)`, and \$ (according to the content of FOLLOW(E)):

State	ACTION						GOTO		
	`id`	`+`	`*`	`(`	`)`	\$	E	T	F
0	s5			s4					
1		s6							
2		r2	s7		r2	r2			
3									
4	s5			s4					
5									
6	s5			s4					
7	s5			s4					
8		s6			s11				
9			s7						
10									
11									

Similarly, for the second item that we underlined: $T \to F\bullet$. Since this item is in I_3, since $T \to F$ is grammar production number 4, and since FOLLOW(T) is {`+`, `*`, `)`, \$}, we put the action "r4" in the row number 3 in the columns for `+`, `*`, `)`, and \$:

State	ACTION						GOTO		
	`id`	`+`	`*`	`(`	`)`	\$	E	T	F
0	s5			s4					
1		s6							
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4					
5									
6	s5			s4					
7	s5			s4					
8		s6			s11				
9			s7						
10									
11									

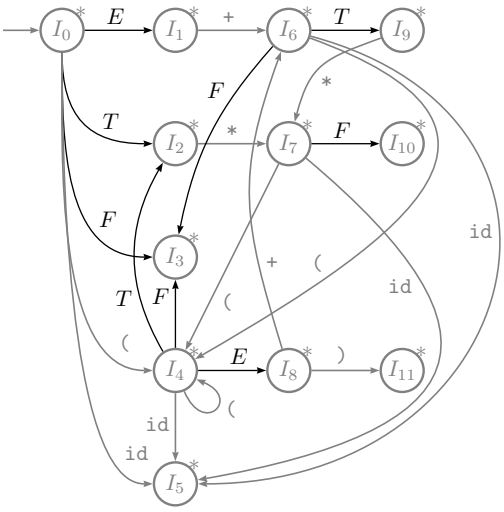
We proceed like this for the four other underlined items and we then get the following parsing table:

State	ACTION						GOTO		
	`id`	`+`	`*`	`(`	`)`	\$	E	T	F
0	s5			s4					
1		s6							
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4					
5		r6	r6		r6	r6			
6	s5			s4					
7	s5			s4					
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

For the third rule above, we have to identify in our sets of items the one item of the form $E' \to E\bullet$, where E' is the start symbol of the augmented grammar and E is the start symbol of the original (non-augmented) grammar. This item is in the set of items I_1. We therefore put an "accept" action in row 1 in the column for \$ (which means the input should be empty at the point in time when the parser accepts the input):

State	ACTION						GOTO		
	`id`	`+`	`*`	`(`	`)`	$	E	T	F
0	s5			s4					
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4					
5		r6	r6		r6	r6			
6	s5			s4					
7	s5			s4					
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Finally, for the fourth rule above, we just have to identify in the DFA all the transitions on nonterminals, which are all the transitions we have not considered before:



For each of these nine transitions, the fourth rule above tells us to put a corresponding goto entry in the table: if there is transition on nonterminal A from I_i to I_j in the DFA then we put "j" in GOTO$[i, A]$:

State	ACTION						GOTO		
	`id`	`+`	`*`	`(`	`)`	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

And we are then done with constructing the parsing table. This table is, unsurprisingly, the same table that we used when we looked at how to use the parsing algorithm to parse some input. Note that all the shift actions in the ACTION part of the table and all the entries in the GOTO part of the table together form the complete transition table for the DFA.

So, to summarize, here are all the steps that are required to create the parsing table for an SLR(1) parser:

- Eliminate any ambiguity from the grammar.
- Augment the grammar with a new start symbol.
- Compute the sets of items for the grammar and build the corresponding DFA.
- Number the grammar productions.
- Compute the FIRST sets for all tokens and nonterminals in the grammar.
- Compute the FOLLOW sets for all nonterminals in the grammar.
- Use the sets of items, the DFA, and the FOLLOW sets to construct the parsing table.

Constructing the LR(0) Parsing Table

In the previous sections we have seen how to construct the parsing table for an SLR(1) parser, which uses a "Simple" parsing table construction method and requires at most one token of lookahead to decide when to do reductions. In this section we are looking quickly at the LR(0) method, which in fact is even simpler and requires no token of lookahead to decide when to do reductions.

As we have indicated above, the SLR(1) parsing table and the LR(0) parsing table are constructed based on the exact same kind of LR(0) items. So to

construct an LR(0) parsing table there is no need to change the sets-of-items construction algorithm, and both SLR(1) and LR(0) parsers are then based on the same DFA.

The difference between the two kinds of parsers is in the way the parsing table is constructed from the DFA. For SLR(1) parsers, we used the following second rule to construct the parsing table:

- if $A \to \alpha\bullet$ is an item in the set of items I_i then set ACTION$[i, \texttt{a}]$ to "reduce $A \to \alpha$" for all `a` in FOLLOW(A) (in practice we will give a number to each grammar production $A \to \alpha$ and put in the table "reduce" followed by the production's number);

In this rule, a reduction occurs only when the next input symbol `a` is in FOLLOW(A). Since, by definition, an LR(0) parser uses zero tokens of lookahead, the rule above should be modified so that the parser can decide whether to do a reduction or not without having to check first what the next input symbol is.

In an LR(0) parser the rule is therefore simplified as follows:

- if $A \to \alpha\bullet$ is an item in the set of items I_i then set ACTION$[i, \texttt{a}]$ to "reduce $A \to \alpha$" for all `a`.

This rule does not use FOLLOW(A) and does a reduction regardless of what the next input symbol `a` might be, so the parser does no longer need to check what `a` is before doing a reduction. In other words, the parser does not need to use a lookahead token anymore.

This makes parsing tables for LR(0) parsers simpler to build than parsing tables for SLR(1) parsers (even though the S in "SLR" stands for "simple"). It means that, when a set of items contains an item with a dot at the right end of the item's RHS, we have to put a reduce action in *all* the columns of the corresponding row in the ACTION part of the table. This is simple to do since we do not have to look at the content of the FOLLOW sets. In fact we do not need to even compute FOLLOW sets anymore.

For example, using the same augmented grammar, sets of items, and DFA as above, the LR(0) parsing table is:

State	ACTION						GOTO		
	`id`	`+`	`*`	`(`	`)`	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2	r2	r2	s7,r2	r2	r2	r2			
3	r4	r4	r4	r4	r4	r4			
4	s5			s4			8	2	3
5	r6	r6	r6	r6	r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9	r1	r1	s7,r1	r1	r1	r1			
10	r3	r3	r3	r3	r3	r3			
11	r5	r5	r5	r5	r5	r5			

This LR(0) table is very similar to the SLR(1) table we computed before, except that each reduce action has been extended to cover all the columns of the ACTION part of the table. This means that the parser does the corresponding reduce action regardless of which token appears next in the input, i.e. the parser does not use a lookahead token anymore to decide whether to reduce or not.

Note that shifts do not involve peeking at the next token in the input either, because, when a shift action has to be done, the parser can always shift the next input token (without peeking at it first), then, once the token is on the stack, look at that new token at the top of the stack and at the state number directly below it to decide which state to associate with the new token on the stack (that information can come from the GOTO part of the parsing table, which can be extended to have columns for tokens, since the GOTO part of the parsing table already describes the same kind of information for nonterminals).

Since reduce actions now cover entire rows and so do not need to use a lookahead token, and since shifts can be done too without such lookahead token, it then becomes useless to have multiple columns in the ACTION part of the table: all the columns in the ACTION part can be merged into a single column and the parser can then decide whether to shift or reduce just by looking at the current state at the top of the stack.

The disadvantage of using LR(0) parsing tables is that, as can be seen in the parsing table above, such tables very often contain conflicts (shift-reduce or reduce-reduce). The table above, for example, contains two shift-reduce conflicts, even though the grammar on which the table is based is very simple.

So LR(0) parsing table are very simple to use but very often contain conflicts, while SLR(1) parsing tables are a bit more complicated to build (you have to compute all the FOLLOW sets) but contain fewer conflicts. This is a very common trade-off when building parsing tables: the fewer conflicts parsing tables have, the more difficult it is to build them.

So in practice people almost never use LR(0) parsing tables, simply because these tables end up having many conflicts. Note though that there is a theorem in the theory of context-free grammars that says that any language that can

be parsed by a bottom-up parser can in fact be parsed using an equivalent LR(0) parsing table with no conflict. This theoretical result is not very useful in practice though, because, while it is possible in theory to use an LR(0) parser to parse any language that can be parsed by any LR parser, doing so requires rewriting the grammar for the language in such a way that the resulting LR(0) parsing table ends up having a huge number of states, which in turn requires a lot of memory in the parser. So in practice transforming a grammar so that an LR(0) parser can be constructed for the corresponding language is never done.

Constructing the LR(1) and LALR(1) Parsing Tables

The SLR(1) parsing tables have fewer conflicts than LR(0) parsing tables, but there are still many unambiguous grammars for which the SLR(1) parsing table has a conflict (of course if the grammar is ambiguous then *all* parsing tables will always have a conflict, regardless of which methods are used to build the tables).

For example, consider the following unambiguous grammar for assignments:

$$\begin{array}{lcl} S & \rightarrow & L = R \\ S & \rightarrow & R \\ L & \rightarrow & \texttt{*}\ R \\ L & \rightarrow & \texttt{id} \\ R & \rightarrow & L \end{array}$$

When given the input `id = id`, the parser can take the following sequence of steps (using items in what looks like a derivation going backwards, to simplify the explanation):

$$\begin{array}{lll} & \bullet\texttt{id = id} & \\ \Rightarrow & \texttt{id}\bullet\texttt{ = id} & \text{shift} \\ \Rightarrow & L\bullet\texttt{ = id} & \text{reduce} \end{array}$$

At this point the parser has the choice between reducing the L into an R using $R \rightarrow L$ or shifting the `=` and `id`, reducing the second `id` into an L and then an R, and then reducing everything using $S \rightarrow L = R$. So there is a shift-reduce conflict. In fact if you define I_2 as goto(I_0, L) and I_6 as goto$(I_2, =)$ (see the Dragon Book page 229 for the complete list of the sets of items) you will see that I_2 contains two items $S \rightarrow L\bullet{=}R$ and $R \rightarrow L\bullet$ and therefore that the SLR(1) parsing table has a "s6,r5" conflict in row 2 for the column for `=`.

The reason for this conflict is not that the grammar is ambiguous (it is not) but that the parsing table construction method for SLR(1) parsing table is not powerful enough. If you look at the grammar, you will realize that an L on the left of an `=` should only be reduced into an R if it is preceded by a `*`:

$$
\begin{array}{lll}
 & \bullet\texttt{*id = id} & \\
\Rightarrow & \texttt{*}\bullet\texttt{id = id} & \text{shift} \\
\Rightarrow & \texttt{*id}\bullet\texttt{ = id} & \text{shift} \\
\Rightarrow & \texttt{*}L\bullet\texttt{ = id} & \text{reduce} \\
\Rightarrow & \texttt{*}R\bullet\texttt{ = id} & \text{reduce} \\
\Rightarrow & L\bullet\texttt{ = id} & \text{reduce} \\
\Rightarrow & L\texttt{ = }\bullet\texttt{id} & \text{shift} \\
\Rightarrow & L\texttt{ = id}\bullet & \text{shift} \\
\Rightarrow & L\texttt{ = }L\bullet & \text{reduce} \\
\Rightarrow & L\texttt{ = }R\bullet & \text{reduce} \\
\Rightarrow & S\bullet & \text{reduce}
\end{array}
$$

while if an L on the left of an `=` is not preceded by a `*` then a shift should be done instead:

$$
\begin{array}{lll}
 & \bullet\texttt{id = id} & \\
\Rightarrow & \texttt{id}\bullet\texttt{ = id} & \text{shift} \\
\Rightarrow & L\bullet\texttt{ = id} & \text{reduce} \\
\Rightarrow & L\texttt{ = }\bullet\texttt{id} & \text{shift} \\
\Rightarrow & L\texttt{ = id}\bullet & \text{shift} \\
\Rightarrow & L\texttt{ = }L\bullet & \text{reduce} \\
\Rightarrow & L\texttt{ = }R\bullet & \text{reduce} \\
\Rightarrow & S\bullet & \text{reduce}
\end{array}
$$

The problem is that the method we use to construct SLR(1) parsing tables decides to do a reduction whenever the set of items corresponding to the current state of the parser contains an item of the form $A \to \alpha\bullet$ and the next token `a` in the input is in FOLLOW(A). If the stack has the shape $\beta\alpha$ before the reduction, the stack has then the shape βA after the reduction. But there are cases where the βA on the stack cannot be followed by the token `a` in the input, even though `a` is in FOLLOW(A)! In the example above, R can be followed by `=`, but only if there is a `*` below R in the stack (i.e. only if β is `*`). If there is nothing below R in the stack (β is ε) then R cannot be followed by `=`, even though `=` is in FOLLOW(R). Put another way, whenever the set of items corresponding to the current state of the parser contains an item of the form $A \to \alpha\bullet$, the reduction should happen only for *some* tokens `a` in FOLLOW(A), not for *all* of them. By deciding to reduce the stack for all `a` in FOLLOW(A), the SLR(1) parsing table construction method is sometimes too eager to reduce, which leads to shift-reduce conflicts.

So SLR(1) parsing tables do not always work. In fact there are many cases in practice where SLR(1) parsing tables end up having conflicts. The solution to this problem is to use a more powerful kind of LR parsers: LR(1). Such parsers do not use LR(0) items but LR(1) items which include more information: an LR(1) item is a grammar production with a dot somewhere in it *plus* an extra symbol (token or \$) of lookahead. So an LR(1) item looks like this: "$A \to X\bullet YZ$, `a`". The extra `a` in the item normally plays no role, except when the parser has to decide whether to reduce or not. If an LR(1) item is of the

form "$A \to \alpha\bullet$, a" and α is currently at the top of the stack, then the parser only does a reduction using $A \to \alpha$ if the next token in the input actually is a. Note that, in such a case, the a will always be in FOLLOW(A) but in some cases there are some a in FOLLOW(A) for which the parser will end up *not* doing a reduction, so this method creates a parsing table with fewer reductions than an SLR(1) table, which leads to fewer conflicts. In other words, because LR(1) items (used by the LR(1) parsing table construction method) contain more information than the LR(0) items (used by the LR(0) and SLR(1) parsing table construction methods), an LR(1) parser can make more precise decisions than an SLR(1) parser, which means an LR(1) parser will be able to handle more grammars and more programming languages.

For example, for the grammar above, state 2 of the SLR(1) parser contains the LR(0) items $S \to L\bullet{=}R$ and $R \to L\bullet$, so there is a shift-reduce conflict. State 2 of the LR(1) parser contains the LR(1) items $S \to L\bullet{=}R, \$$ and $R \to L\bullet, \$$, so the reduction can only occur if the end of the input has been reached. If the current L on the stack is followed by = in the input then only the shift action can take place, not the reduction, and therefore there is no conflict for the LR(1) parser.

A side effect of using LR(1) items is that some states from an SLR(1) table will be split into multiple states in the corresponding LR(1) table, so LR(1) tables usually are much bigger than SLR(1) tables. For example, for the grammar above, state 8 of the SLR(1) parser, which contains the single LR(0) item $R \to L\bullet$, will be split into two separate states of the LR(1) parser, one state containing the LR(1) item $R \to L\bullet, =$ and another state containing the LR(1) item $R \to L\bullet, \$$. Another difference is that the DFA path leading to the first state will not have a transition on = before reaching the state, while the DFA path leading to the second state will have a transition on =. In other words, the first state will be used to reduce L into R only on the left of an = token while the second state will be used to reduce L into R only on the right of an = token. In the DFA for the SLR(1) parser, both paths lead to the same state 8 and when the parser is in state 8 it does not take into account which path was used to reach state 8 to make a decision on what to do next (i.e. when in state 8, the parser only cares about the L at the top of the stack to decide what to do next, it does not care about whether there might be a = token somewhere below the L in the stack or not). The DFA for the LR(1) parser is therefore more precise than the one for the SLR(1) parser, but this higher precision comes at the price of a bigger parsing table.

Of course there are grammars for which even LR(1) is not sufficient. The same idea can then be used to construct LR(2) parsing tables (which have two tokens of lookahead in each LR(2) items), LR(3) parsing tables, etc. Each one will result in parsing tables which are more precise and therefore have fewer conflicts than the previous ones, which means the corresponding parsers will be able to handle more and more context-free grammars. Of course the corresponding parsing tables will then be bigger and bigger.

Note though that there are context-free grammars which cannot be parsed using LR(k) parsing tables for any k. This includes obviously all ambiguous

context-free grammars, but there are also unambiguous context-free grammars which will create conflicts in LR(k) parsing tables for all possible values of k. This means that LR(k) does not completely cover the set of context-free grammars, but the difference is unimportant since in practice those kinds of non-LR(k) grammars are never found when parsing programming languages.

Even for languages that can be parsed using LR(k) parsing tables (which is most languages) there are still two problems with the parsing tables (including LR(1) tables):

- They become more and more complicated to build. This is not too much of a problem since building the parsing tables can always be done using some automated tool.
- More importantly, the extra tokens of lookahead in the LR(k) items lead to many states being split. This results in enormous parsing tables with thousands of rows which require a lot of memory. For $k \geq 2$, the number of columns becomes enormous too since there has then to be one column for each possible combination of k input symbols, i.e. an exponential number of them.

Because of these problems (and in particular because of the second one) people do not usually use LR(1), LR(2), etc., parsers. What they use instead are LALR(1) parsers.

An LALR(1) parsing table is constructed essentially in the same way as an LR(1) parsing table, using LR(1) items. The difference is that, once all the sets of LR(1) items have been constructed, two sets which contain items with the same RHS but different tokens of lookahead (for example, one set containing the item "$A \rightarrow X \bullet YZ$,a" and one set containing the item "$A \rightarrow X \bullet YZ$,b") are merged together into a single set of items (which then contains an item like "$A \rightarrow X \bullet YZ$,a|b"). This merging undoes the splitting that was done when moving from SLR(1) to LR(1) parsers, but the merging is done in such a manner that LALR(1) parsers lose very little power compared to LR(1) parsers.

For example, for the grammar above, state 8 of the SLR(1) parser contains the single LR(0) item $R \rightarrow L \bullet$, and the LR(1) parser contains two corresponding states, one for the LR(1) item $R \rightarrow L \bullet, =$ and another for the LR(1) item $R \rightarrow L \bullet, \$$. The LALR(1) parser then contains a single state (just like the SLR(1) parser) that contains the LALR(1) item $R \rightarrow L \bullet, = | \$$, which is the result of merging the two corresponding states of the LR(1) parser.

Another way to look at it is to say that, if the LR(1) parsing table looks like this:

State	ACTION					
i	...	r3	r3	...		...
...	...					
j	...			...	r3	...

then in the LALR(1) parsing table the two rows for i and j are merged:

State	ACTION					
i|j	...	r3	r3	...	r3	...
...	...					

Since in practice many states share similar-looking items (because many LR(1) items are the same except for the lookahead token) doing this results in massive compression of the parsing table. In fact an LALR(1) parsing table is guaranteed to have exactly the same number of rows as an SLR(1) parsing table (and the same number of columns, of course, since both use only one token of lookahead). The result is that an LALR(1) parsing table is typically an order of magnitude smaller than the corresponding LR(1) parsing table.

Of course doing this compression results in some loss of precision in the LALR(1) parsing table, compared to the corresponding LR(1) parsing table. In practice (again) the loss of precision is minimal, and LALR(1) parser can handle the vast majority of programming language constructs we might be interested in. LR(1) parsers can handle those constructs too and a few more, but this extra power is of little use in practice and the LR(1) parsing tables are huge. SLR(1) parsing tables are of the exact same size as LALR(1) parsing tables, but in practice they contain many conflicts, which LALR(1) parsing tables do not (the difference being that the LALR(1) tables are more precise about when to reduce the content of the stack, compared to SLR(1) tables). The result is that LALR(1) parsing tables are the kind of parsing tables that are usually used by people, because they nicely combine the small size of SLR(1) parsing tables with much of the power of LR(1) parsing tables. This explains why most tools like YACC or Bison which are used to generate bottom-up parsers automatically in fact generate LALR(1) parsers, simply because these parsers use fairly small parsing tables while being able to handle most programming language constructs.

Error Recovery For LR Parsers

An LR parser detects a syntax error when it tries to use an empty entry in the ACTION part of its parsing table. The GOTO part of the parsing table is only used after a reduction, and the way we constructed the GOTO part of the table based on a DFA guarantees that there can never be any error when using that part of the table.

Once a syntax error has been discovered, the parser should print an appropriate error message. One nice property of LR parsers is that they never shift any erroneous token from the input onto the stack, so when a syntax error is discovered, the cause of the syntax error is always guaranteed to be the next token in the input, which makes it easier to print an error message. Once the error message has been printed, the parser has to try to recover from the error to try to discover more syntax errors in the input. This is then done as follows.

First, for each row in the ACTION part of the parsing table, we find a nonterminal that we are currently in the middle of parsing. We do this by looking at the items in the set of items corresponding to the row. For example,

if we look again at the SLR(1) parsing table that we constructed for our grammar for simple arithmetic expressions:

State	ACTION						GOTO		
	`id`	`+`	`*`	`(`	`)`	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

we have row 7 which is associated with the set of items I_7 which is $\{T \to T\texttt{*}\bullet F,\ F \to \bullet\texttt{(}E\texttt{)},\ F \to \bullet\texttt{id}\}$. This set of items means that the parser is currently in the middle of parsing a T, that it has already matched the $T\texttt{*}$ part of that T, and that it expects to match next an F in the input.

So if there is a syntax error while the parser is in state 7 (i.e. state 7 is in the pair at the top of the stack) then it means that $T\texttt{*}$ is currently in the two pairs at the top of the stack (with state 7 associated with `*` in the topmost pair) and it also means that the next token in the input is either `+`, `*`, `)`, or $, since these are the four columns with blank entries in row number 7 of the ACTION part of the parsing table. In essence this means that `+`, `*`, `)`, and $ cannot start the F that the parser is trying to match next in the input (in fact an F can only start with `id` or `(`, as indicated by the other items in I_7). What the parser can do then is remove $T\texttt{*}$ from the stack and remove tokens in the input until it finds in the input a token that was supposed to follow the T that the parser was in the middle of matching. By doing this, the parser forgets the part of T it has already matched and placed on the stack (the $T\texttt{*}$ part) and forgets the tokens in the input that were supposed to match the remaining F that the parser was looking for next in the input. Then the parser can just pretend that the whole T was actually matched (even though is was not) by directly putting a new T at the top of the stack and using the GOTO part of the parsing table to find the associated state, as if the new T were the result of doing a reduction of $T\texttt{*}F$ into the T.

In general, if the parser is currently in the middle of parsing a nonterminal A based on an item like $A \to \alpha\bullet\beta$ then, if a syntax error occurs, we know that we have α currently at the top of the stack, and that the next token in the input is not in FIRST(β). What the parser does then is to remove α from the stack and skip tokens in the input until it finds a token in the input which is in FOLLOW(A). This corresponds to removing from the stack the part α of A the

parser has already matched, and removing from the input all the tokens that were supposed to match the β part of A that remained to be matched, even though that match turned out not to be possible because of the syntax error. To determine were the β was supposed to end in the input, the parser skips input tokens until it sees in the input a token that is supposed to follow the A that the parser was trying to match. Once it has removed α from the stack and removed the input tokens that should have been the β part of the A, the parser can just pretend that the A was actually matched by directly putting a new A at the top of the stack and using the GOTO part of the parsing table to find the associated state, as if the new A were the result of doing a reduction of $\alpha\beta$ into the A.

Let's try for example to parse the following input: "`id + id * *`", using our SLR(1) parsing table above for our usual grammar for simple arithmetic expressions. Here is what it looks like:

Stack	Input	Output
0	`id + id * * $`	
0 $\underbrace{\texttt{id}\ 5}$	`+ id * * $`	shift 5
0 $\underbrace{F\ 3}$	`+ id * * $`	reduce $F \to$ `id`
0 $\underbrace{T\ 2}$	`+ id * * $`	reduce $T \to F$
0 $\underbrace{E\ 1}$	`+ id * * $`	reduce $E \to T$
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}$	`id * * $`	shift 6
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{\texttt{id}\ 5}$	`* * $`	shift 5
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{F\ 3}$	`* * $`	reduce $F \to$ `id`
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{T\ 9}$	`* * $`	reduce $T \to F$
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{T\ 9}\ \underbrace{\texttt{*}\ 7}$	`* $`	shift 7
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{T\ 9}\ \underbrace{\texttt{*}\ 7}$	`* $`	syntax error
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}$	`* $`	remove α from stack
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}$	`* $`	remove β from input
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{T\ 9}$	`* $`	fake reduction of T
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{T\ 9}\ \underbrace{\texttt{*}\ 7}$	`$`	shift 7
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{T\ 9}\ \underbrace{\texttt{*}\ 7}$	`$`	syntax error
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}$	`$`	remove α from stack
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}$	`$`	remove β from input
0 $\underbrace{E\ 1}\ \underbrace{\texttt{+}\ 6}\ \underbrace{T\ 9}$	`$`	fake reduction of T
0 $\underbrace{E\ 1}$	`$`	reduce $E \to E$ `+` T
0 $\underbrace{E\ 1}$	`$`	accept

Here everything works as usual until the parser encounters the second `*` in the input. At that point in time the parser is in state 7, as indicated by the

state number at the top of the stack. The parser is therefore in the middle of matching a T, has already matched the T* part of this T, and is looking for an F next in the input (all this is indicated by the items in the set of items I_7, in particular by the item $T \rightarrow T*\bullet F$). The parser then finds next in the input a token * that cannot be the start of the F it is looking for (i.e. * is not in FIRST(F)). The parser therefore has detected a syntax error (i.e. the entry for row 7 and column * in the SLR(1) parsing table above is blank). The parser then removes from the stack the α part of the T it has already matched (i.e. it removes from the top of the stack the two pairs for the grammar symbols T* that appear on the left of the dot in the first item $T \rightarrow T*\bullet F$ of I_7). The parser then removes from the input as many tokens as necessary until it finds in the input a token that can follow the T that the parser was trying to match. The set FOLLOW(T) is {+, *,), \$} so the * that is next in the input is an appropriate token to follow a T (that token cannot start the F that was supposed to be part of the T though, hence the syntax error) and therefore nothing is removed from the input (i.e. β is the empty string). The parser then puts on the stack a new T and pretends that this new T is the result of some reduction by using the GOTO part of the table to compute the associated state 9 (i.e. GOTO[6, T] is 9).

After that the parser resumes parsing as usual, shifting the next input token * onto the stack. After the shift the parser is back in state 7 and the next symbol in the input is \$. Since ACTION[7, \$] is empty a second syntax error is therefore detected. Note that here the parser detects a second syntax error but the input only contained one error. This second error is only a consequence of the fact that the recovery from the first error was not perfect. As we have said before, error recovery is only about the parser trying to guess what to do next to recover from an error, but the parser cannot always guess what is the best way to recover.

At the point where this second error is detected, the parser is in state 7, just like when the first syntax error was detected (this is just a coincidence). This means that the parser is again in the middle of trying to match a T: the parser has already matched the T* part and put it on the stack, and was expecting to see next an F (starting either with an `id` or an `(`). To recover from the error the parser therefore again first removes from the top of the stack the T* it has already matched. Next the parser removes tokens in the input until it sees in the input a symbol which is in FOLLOW(T). The set FOLLOW(T) is {+, *,), \$}, which include \$, so the parser does not remove anything from the input (there is nothing left to remove anyway!) Next the parser puts on the stack a new T and again pretends that this new T is the result of some reduction by using the GOTO part of the table to compute the associated state, which is state 9 again.

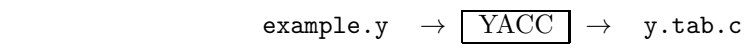
After that the parser resumes once again parsing as usual. The content of the stack is reduced into a single E, the current state of the parser becomes 1, the next input symbol is \$, and the parser therefore accepts the input, after having identified two syntax errors (the second one being only the result of the error recovery not being perfect).

The YACC Parser Generator

As you have seen in the homework assignment, YACC (or its GNU twin called Bison) is a tool to automatically generate the C code for parsers.

The user writes in a "`.y`" specification file the context-free grammar (LALR(1) grammar, to be more precise) that describes the different nonterminals to be identified. The user also writes an action (C code) associated with each grammar production.

The user then runs YACC on this specification file and from this specification file YACC produces another file `y.tab.c` that contains C code that implements a complete parser for the context-free grammar:



The actions that were written in the specification file are copied as is in the file `y.tab.c` at the right places so that the C code of each action is executed every time the corresponding grammar production is used in a reduction.

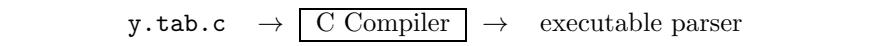
The "`.y`" specification file also has two sections at the beginning and at the end of it where the user can write C declarations and C auxiliary functions that are required by the various actions associated with the grammar productions. These C declarations and auxiliary functions are then copied as is in the C code of the parser generated by YACC. This C code should be enclosed within special braces ("`%{`" and "`%}`") in the first section, because that section can also contain special declarations (like the "`%token`" declaration to define the list of tokens used in the grammar, or some other declaration for operator precedence and associativity that we talk about below).

What YACC does is to read the context-free grammar defined in the "`.y`" file, and transform it into C code using the method we have explained above (computing the sets of LR(1) items and the corresponding DFA, then constructing the LALR(1) parsing table based on the sets and the DFA). So YACC internally implements all the different algorithms we have seen above (the LALR(1) version of them, to be more precise) for you! This is why using tools like YACC or Bison makes writing parsers much easier. In fact very few people (except students...) write parsers themselves, most people just use tools like YACC to automatically generate correct and efficient parsers from simple specifications. The only exception is for LL(1) grammars, in which case people might directly write a simple top-down recursive predictive parser, but otherwise people will very often just use tools to generate the parsers for them.

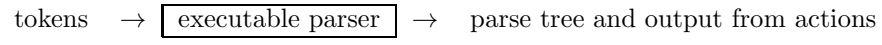
Note that today there are more and more automated tools to create LL(k) parsers (or even something called LL(*) parsers) so not all the tools you might find for various programming languages will generate the same kind of LALR(1) parsers as YACC does. YACC is an acronym for "Yet Another Compiler Compiler", which also tells you that there are many other tools like YACC available. YACC is historically the first such tool that became widely popular though, and it is still today a standard tool for writing parsers.

Once YACC has created the C code for a parser, that parser can then be compiled using a normal C compiler to get an executable program that imple-

ments the parser as well as all the associated C actions that were specified in the original "`.y`" file:



This executable parser can then be run by a computer and given some tokens (presumably coming from a lexer which has been linked together with the parser). The executable parser will then try to match these tokens against the grammar productions (using C code that implements the LR parsing algorithm and uses a table which is the result of transforming the grammar productions into an LALR(1) parsing table) and will execute the corresponding action every time a match is detected:



Three important things need to be noted here:

- YACC itself is not a parser, it is a *parser generator*. YACC is a tool that you can use to create parsers, but it is not itself a parser. Do not confuse YACC with the parsers it creates.

- YACC creates parsers that are implemented in C. This means the parsers generated by YACC can only be executed after they have been compiled by a C compiler. If you need to create a parser that has to be implemented in Java, for example, then you cannot use YACC, you will instead have to use another tool that works very much like YACC but generates parsers that are implemented in Java. In practice there are tools like YACC available for pretty much all the programming languages you can think of. So if you want to generate automatically a parser that is implemented in your favorite programming language, you just have to find the right parser generator for that programming language, write a specification file for your parser, and then apply the tool to the specification file to get the code of a parser written in your favorite programming language.

- The programming language in which the parser is implemented is completely unrelated to the programming language that is processed by the executable parser as input. For example, using YACC you can only generate parsers that are written in C, but these parsers themselves can be parsers for any input language you want. If you remember your homework assignment, the parsers you generated using YACC were all written in C (since this is what YACC does) but the parsers themselves were parsers for programming languages that contained things like "`target temperature 10`", or "`heat off`", or "`zone "foo/bar" {type hint; file "/foo/foo/"};`", etc. So do not confuse the programming language which is used to implement the parser (which is the programming language that should then be supported by the compiler you use to compile the generated parser) with the programming language that is processed by the parser as input.

Resolving Ambiguities With YACC

YACC (and other parser generators like it) has some nice features to help you deal with ambiguous grammars. For example the following grammar:

$$\begin{array}{lcl} E & \rightarrow & E \texttt{ + } E \\ E & \rightarrow & E \texttt{ * } E \\ E & \rightarrow & \texttt{(} E \texttt{)} \\ E & \rightarrow & \texttt{id} \end{array}$$

is very easy to understand but unfortunately it is ambiguous, so it will result in an ambiguous parsing table for all the LR parsing methods (in fact for all parsing methods ever). The following grammar is not ambiguous (it completely specifies the associativity and precedence of `+` and `*`):

$$\begin{array}{lcl} E & \rightarrow & E \texttt{ + } T \\ E & \rightarrow & T \\ T & \rightarrow & T \texttt{ * } F \\ T & \rightarrow & F \\ F & \rightarrow & \texttt{(} E \texttt{)} \\ F & \rightarrow & \texttt{id} \end{array}$$

but it is a lot more difficult to understand. It is also somewhat complicated to modify the first grammar into the second one manually if one wants to use an automated tool to generate a parser starting from the first grammar.

Fortunately YACC has special "`%left`", "`%right`", and "`%nonassoc`" declarations that we can use to manually specify the associativity (or non-associativity) and precedence of tokens without having to modify an ambiguous grammar.

For example, we can write a "`.y`" file containing the ambiguous grammar just above. If we then run YACC on that file we will get errors about various shift-reduce conflicts. If we then add in the first section of the "`.y`" file the following declarations:

```
%left +
%left *
```

YACC will be able to use these declarations to automatically resolve all the conflicts in the grammar for you! What these two declarations do is to tell YACC that `+` and `*` are both left-associative. Since the declaration for `*` appears after the one for `+`, this also tells YACC that `*` has higher precedence than `+`. YACC then generates the LALR(1) parsing table for the ambiguous grammar. This parsing table obviously contains conflicts. YACC then uses the two declarations above to resolve each conflict in the table and ensure that each entry in the parsing table has at most one action in it.

Using such declarations one can have the best of both worlds: use a tool to generate parsers automatically, and at the same time use ambiguous grammars which are simple to read, write, and understand. The only thing that has to be done for this to work is to tell YACC about the associativity and precedence of the various tokens in the grammar (in fact doing it just for the tokens that

represent the various operators in the programming language is often enough to solve all the conflicts).

In fact in some cases it is not even necessary to use declarations: if a grammar is ambiguous and generates shift-reduce conflicts, and YACC does not know how to resolve the conflict because no declaration has been provided to do so, YACC will then automatically generate a parser that always does a shift instead of a reduce. In practice this is often (but not always) the right choice, so in many cases the user can simply rely on the default behavior of YACC and of the generated parser to resolve the problem. In the case of reduce-reduce conflicts, YACC will by default automatically generate a parser that always uses the first grammar production involved in the conflict to reduce the content of the stack. In such a case it is often not clear whether this is the right thing to do, so users are usually advised not to rely on YACC's default behavior and resolve the reduce-reduce conflict manually (as we have said before, reduce-reduce conflicts are usually a good sign that something is actually very wrong about the grammar). In all cases YACC will give a warning when there is a conflict.

Another nice feature of tools like YACC that helps to resolve ambiguities is that they can give you complete debugging information about how the generated parser works. For example YACC has a `-v` option that will generate a file `y.output` that contains a description of all the grammar productions, tokens, parser states, corresponding sets of items, shift and reduce actions from the ACTION part of the table, as well as the GOTO part of the table, that is used by the parser. YACC is able to tell you were in the parsing table there are shift-reduce or reduce-reduce conflicts, and which grammar productions are involved. Using this information, the user can therefore always debug his ambiguous grammar, locate which parts of the grammar create the conflicts (though in practice this often requires some experience and some work...), and then resolve the conflicts, either by using declarations like "`%left`" and "`%right`", or by rewriting the grammar to make it unambiguous if necessary.

If more information is required, then the user can even use YACC's `-t` option. The generated parser will then automatically output a complete trace of what it is doing when given some tokens as input, and the user will then be able to follow and check step by step all the shift and reduce actions done by the parser. This is a good way to identify places where the parser does a shift when the user expected a reduce, or vice-versa.

Interfacing Lex With YACC, Semantic Values

Lexers generated by Lex and parsers generated by YACC are designed to work together. It is usually easy to use a Lex-generated lexer on its own without a parser but YACC-generated parsers are almost always used in conjunction with a Lex-generated lexer.

One basic reason is that using a parser without a lexer is difficult, since there has to be somewhere some piece of code that analyzes the input characters and creates tokens for the parser. It is possible to create a parser that uses individual

input characters directly as tokens, but to do that one has to basically implement all the regular expressions from the lexer as part of the context-free grammar of the parser in order to merge the lexer and the parser together, and as we have discussed before this results in an enormous (and ugly) context-free grammar.

The other reason is that YACC-generated parsers assume many things about the way the lexer behaves, in order to simplify the parser and lexer generation. So, while it is possible to use a YACC-generated parser with a hand-written lexer, the hand-written lexer will have to have an interface that looks very much like the one for a Lex-generated lexer. So in practice people simply directly use a Lex-generated lexer rather than write a lexer by hand.

There are two things in particular that makes the interaction between Lex-generated lexers and YACC-generated parsers easy. One is the "`%token`" declaration that the user puts in his "`.y`" file. This declaration is used by YACC to know which tokens it might receive from the lexer. In fact what YACC does with this declaration is to generate C macros that define integer values for all these tokens. These macros are then placed by YACC into a file called "`y.tab.h`" that the code of the Lex-generated lexer can then use. This ensures that the lexer and parsers always use the same integer values to represent the same tokens, thereby ensuring that the two different phases work in a coherent manner. The "`%token`" declaration also allows YACC to check that the user is not using in his grammar tokens which have not been defined and which are therefore unknown to the lexer.

The second thing that makes the interaction of a Lex-generated lexer and a YACC-generated parser easy is the `yylval` global variable. This variable is shared by the lexer and parser and the lexer can use it to store in it a semantic value associated with a token. The parser can then access this semantic value by using pseudo-variables like `$1`, `$2`, etc., in the C code associated with each grammar production.

What happens is that, in reality, the parser generated by YACC does not use pairs of grammar symbols and state numbers on its stack, but instead uses triples where each grammar symbol and state is also associated with a semantic value. When the parser does a shift action, the semantic value associated with the token is copied from `yylval` into the new triple onto the stack. During a reduction, the C action associated with the grammar production used in the reduction can use the various `$1`, `$2`, etc., pseudo-variables to refer to the various semantic values in the various triples on the stack.

For example, consider again the following grammar for simple arithmetic expressions, with next to each production the associated YACC-like action:

E	$\rightarrow$	E `+` T	`$$ = $1 + $3`
E	$\rightarrow$	T	`$$ = $1`
T	$\rightarrow$	T `*` F	`$$ = $1 * $3`
T	$\rightarrow$	F	`$$ = $1`
F	$\rightarrow$	`(` E `)`	`$$ = $2`
F	$\rightarrow$	`NUM`	`$$ = $1`

Here we have replaced `id` with `NUM` so our example is easier to understand. The

SLR(1) parsing table is therefore the same one as before, except that `id` has been replaced with `NUM`:

State	ACTION						GOTO		
	NUM	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

We assume that the specification of the lexer looks something like this:

```
[0-9]+      yylval = atoi(yytext); return NUM;
\+          return *yytext;
[ \t]+      /* ignore whitespace */;
```

Let's look then at what happens when the compiler is given the following input: `13 + 27`. The parser starts in the following state:

Stack	Next Token	`yylval`	Action
0			

The parser then calls the lexer to get the first input token. The lexer reads the characters "`13`", matches them with the regular expression "`[0-9]+`", converts the two characters "`13`" of the lexeme found in `yytext` into the integer 13, puts this value 13 into `yylval`, and returns the token `NUM` to the parser:

Stack	Next Token	`yylval`	Action
0	NUM	13	

Based on the state at the top of the stack and the next token in the input, the parser then shifts the token onto the stack. The parsing table tells the parser that the associated state is 5, and the associated semantic value 13 is copied from `yylval`, all into a single triple:

Stack	Next Token	`yylval`	Action
0 NUM 13 5		13	shift 5

The parser then needs to know what the next input token is, so it calls the lexer again, the lexer reads the character `+`, matches it with the regular expression "`\+`", does not modify `yylval`, and returns a token to the parser which is the ASCII code for the character `+`:

Stack	Next Token	yylval	Action
0 NUM 13 5	+	13	

The parser is then in state 5 and the next input token is +. The parser therefore uses grammar production 6 to reduce the NUM into an F. As it does this, it gets the value of the new state for the new triple at the top of the stack from the GOTO part of the table (i.e. GOTO$[0, F]$ is 3) and it uses the semantic action "$$ = $1" for production 6 to compute the new semantic value for the new triple. This action says to take the semantic value associated with NUM (represented by $1) and use that value as the semantic value for the new triple (represented by $$):

Stack	Next Token	yylval	Action
0 F 13 3	+	13	reduce $F \rightarrow$ NUM

Then the parser reduces the F into T, and then T into E. The semantic action for these two reductions is still "$$ = $1" so the semantic value is always the same:

Stack	Next Token	yylval	Action
0 T 13 2	+	13	reduce $T \rightarrow F$
0 E 13 1	+	13	reduce $E \rightarrow T$

In state 1 with + as the next input token, the parser has to shift. Here the semantic value is again copied from yylval. Note here that the lexer never assigned a new semantic value to yylval before returning the token +, so the semantic value associated with + is in fact the old semantic value associated with the first token NUM. This does not matter since we are going to see that the semantic value associated with + is never used:

Stack	Next Token	yylval	Action
0 E 13 1 + 13 6		13	shift 6

Then the parser needs another input token. Again the parser calls the lexer, which reads the characters "27", matches them with the regular expression "[0-9]+", converts the two characters "27" of the lexeme found in yytext into the integer 27, puts this value 27 into yylval, and returns the token NUM to the parser:

Stack	Next Token	yylval	Action
0 E 13 1 + 13 6	NUM	27	

The parser then shift the token NUM and copies at the same time the value of yylval onto the stack:

Stack	Next Token	yylval	Action
0 E 13 1 + 13 6 NUM 27 5		27	shift 5

Next the parser calls the lexer again to get the next input symbol, the lexer then detects that the end of the input has been reached, and returns the special symbol $ to the parser (without changing the value of `yylval` first, but it does not matter since that value is not used):

Stack	Next Token	`yylval`	Action
0 $\underbrace{E\ 13\ 1}$ $\underbrace{\texttt{+}\ 13\ 6}$ $\underbrace{\texttt{NUM}\ 27\ 5}$	$	27	shift 5

As before the parser then reduces the `NUM` at the top of the stack first into an F and then into a T. These reductions use the semantic action "`$$ = $1`" so the semantic value of the triple at the top of the stack does not change:

Stack	Next Token	`yylval`	Action
0 $\underbrace{E\ 13\ 1}$ $\underbrace{\texttt{+}\ 13\ 6}$ $\underbrace{F\ 27\ 3}$	$	27	reduce $F \to$ `NUM`
0 $\underbrace{E\ 13\ 1}$ $\underbrace{\texttt{+}\ 13\ 6}$ $\underbrace{T\ 27\ 9}$	$	27	reduce $T \to F$

Next the parser has to use the grammar production $E \to E$ `+` T to reduce the three triples at the top of the stack into a single one. The corresponding semantic action is "`$$ = $1 + $3`". Here `$1` represent the semantic value in the triple associated with the first grammar symbol in the RHS of the grammar production: the triple for E, so the bottommost triple in the stack. The pseudo-variable `$1` therefore represents the value 13. The pseudo-variable `$3` represents the semantic value in the triple associated with the third grammar symbol in the RHS of the grammar production: the triple for T, so the topmost triple in the stack. The pseudo-variable `$3` therefore represents the value 27. The action then adds 13 and 27 and store the result in `$$`, which represents the semantic value for the new triple in the stack:

Stack	Next Token	`yylval`	Action
0 $\underbrace{E\ 40\ 1}$	$	27	reduce $E \to E$ `+` T

Note that the semantic value associated with `+`, and represented by the pseudo-variable `$2`, was not used in the semantic action, and therefore it did not matter what that value was. Therefore it did not matter that the lexer never assigned a value to `yylval` before returning the token `+` to the parser.

The parser then accepts, as indicated by the parsing table:

Stack	Next Token	`yylval`	Action
0 $\underbrace{E\ 40\ 1}$	$	27	accept

and the semantic value associated with the start symbol is then 40, which is the result of the addition `13 + 27`. So what the semantic actions associated with the various grammar productions actually did was to compute the semantic value associated with the whole expression, as it was parsed. This shows that an LR parser can easily be used to compute semantic values, as we will see in the next section.

To finish there are a few things to note here:

- Here we have used integers for the semantic value, but it is possible in the YACC "`.y`" specification file to redefine the type `YYSTYPE` of `yylval` to be anything we want. In particular it is possible to use `yylval` and all the associated `$1`, `$2`, ..., `$$` pseudo-variables to represent parse trees. Note that a parser generated by YACC does not automatically build a parse tree for you: if you want a parse tree you have to construct it yourself using semantic actions. The `$$` pseudo-variable will then represent the current parse tree and each `$1`, `$2`, etc., pseudo-variable will then represent a sub-tree in the current parse tree.

- If no semantic action is specified for a grammar production, then the default semantic action of the parser will be "`$$ = $1`". For simple grammar productions (like $F \rightarrow$ `NUM`) this is often enough.

- As we have said before, an LR parser does not in fact need to have the various grammar symbols on its stack, since the states in the stack contain enough information for the parser to know what it has done so far. So in practice a parser generated by YACC only uses pairs on its stack, and each pair contains just a state number and the associated semantic value.

- Instead of thinking of a single stack with pairs of states and semantics values, one can equivalently think of having two stacks that always change in unison: one stack that only contains state numbers, and one stack that only contains the associated semantic values. The result is the same.

This now completes our overview of syntax analysis. The next phase in a compiler is then semantic analysis.