Syntax Analysis

Read Chapter 4 of the Dragon Book (you can skip section 4.6) or Chapter 3 of the Mogensen book.

Introduction

Syntax analysis is the second phase of a compiler. The syntax analyzer (parser) takes the stream of tokens generated by the lexer from the source program and checks this stream of tokens against a grammar that describes the expected structure of all syntactically valid programs that can be written in the programming language that the compiler accepts as input. In essence the job of the parser is to check that the tokens received from the lexer are in an order which is consistent with the structure of the programming language. If the order of the tokens is correct then the parser creates a parse tree that explicitly represents the structure of the source program. If the order of the tokens is incorrect then the parser generates a syntax error and tries to do some error recovery.

The grammar used by the parser to check the structure of source programs is very often a *context-free grammar* (CFG). There exists other types of grammars which are more powerful (can describe more complex programming languages) than CFGs but CFGs are very popular for describing the structure of programming languages because:

- in practice the syntax of the vast majority of programming language constructs most programmers care about can be easily and precisely described by CFGs, so there is no real need for the syntax of programming languages to be based on more powerful grammars;
- it is possible to create automatically efficient parsers for many (but not all) CFGs, as we are going to see in these lecture notes, while parsers for more powerful grammars tend to be very inefficient;
- the algorithms we are going to use to create these efficient parsers have the extra benefit of being able to automatically tell us when they think a grammar is ambiguous and why (we want to avoid using ambiguous grammars because then a source program might not have a unique parse tree).

We will use the *Backus-Naur Form* (BNF) to write grammars, as this notation is easy to understand.

There are two main types of efficient parsers: *top-down* and *bottom-up*, depending on the order in which they build parse trees. Top-down parsers build parse trees starting from the root at the top of the parse tree and going down towards the leaves, while bottom-up parsers build parse trees starting from the leaves at the bottom of the parse tree and going up towards the root.

Efficient parsers cannot be created for all CFGs though. Efficient bottom-up parsers can only be constructed for a strict subset of CFGs called the LR(1) grammars. Fortunately LR(1) grammars are still expressive enough to be able to describe the syntax of most programming language constructs we might care about, so in practice we will not lose much by restricting ourselves to LR(1) grammars. We are going to see that creating efficient bottom-up parsers for LR(1) grammars is quite a bit of work though. This is why creating this kind of parsers is usually not done by hand but instead is done using tools (like YACC or Bison) that automatically generate the code of the parsers based on a grammar specification (a `.y` file in the case of YACC/Bison).

Before looking at LR(1) grammars and the bottom-up parsers for them, we will first look at an even more restricted subset of CFGs called LL(1) grammars. LL(1) grammars have the nice property that it is easy to create top-down parsers for them. It is in fact so easy that LL(1) grammars are often used when one wants to write a parser by hand. So LL(1) grammars and their top-down parsers will act for us as a nice introduction to the topic of syntax analysis before looking at LR(1) grammars and their more complex bottom-up parsers.

It turns out that LL(1) grammars are not just a strict subset of CFGs, but are also a strict subset of LR(1) grammars (which are themselves a strict subset of all the CFGs) so LR(1) grammars and their bottom-up parsers are in fact strictly more powerful (can analyze more complex grammars) than LL(1) grammars and their top-down parsers: every LL(1) grammar is also an LR(1) grammar, and for every top-down parser there is an equivalent bottom-up parser, but the opposite is not true. In short, there is nothing that you can do with an LL(1) grammar and a top-down parser that you cannot do with an LR(1) grammar and a bottom-up parser. This means that in practice most programming languages are based on the more powerful LR(1) grammars and compilers for these languages therefore use bottom-up parsers, while the simpler LL(1) grammars are mostly used for simple domain-specific languages (the languages used for the configuration files of various software, for example, or used in simple network protocols like SMTP) for which a simple hand-written top-down parser is good enough. One interesting case is Pascal, which is a general-purpose programming language that was designed right from the start to have a simple LL(1) grammar, which then makes it easy to parse Pascal programs.

Before we start looking in details at the various kinds of parsers though, we are first going to look at a quick overview of CFGs.

Context-Free Grammars

Many programming language constructs have a recursive structure. For example, in the C language, if E_1 and E_2 are arithmetic expressions then E_1 `+` E_2 is an arithmetic expression too. Similarly, if E is an expression and S_1 and S_2 are statements then `if(`E`)` S_1 `else` S_2 is a statement too.

Remember from the previous set of lecture notes that regular expressions cannot be used to describe languages (sets of strings) where the strings have a recursive structure. To do this we need to use a more powerful notation than regular expressions, that allows us to describe the recursive nature of the strings we are interested in recognizing. This is why we need to use context-free grammars. As we have noted above, context-free grammars are not the only kind of grammars that can be used to describe recursive structures as there exists other kinds of more powerful grammars, but in the vast majority of cases context-free grammars will be enough for the kind of constructs we want to describe, so we will not look at the other kinds of grammars in this course.

To write the context-free grammars themselves, we will use a notation called the Backus-Naur Form (BNF). There are other ways to describe context-free grammars (notably one kind of notation based on diagrams) but we will only use BNF in this course. Here is an example of using the BNF notation to write a simple context free grammar for arithmetic expression:

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

In this grammar we have five different *productions*. Each production describes one possible structure for the E *nonterminal*. A nonterminal is simply a grammar symbol that names a particular category (i.e. set) of program strings. In this example, E is simply a grammar symbol that names the category of arithmetic expressions. A production always has a single nonterminal on the left hand side (LHS) of the arrow and a combination of nonterminals and tokens on the right hand side (RHS) of the arrow. So, for example, the first production in the grammar above states that one possible form for an arithmetic expression is to have an arithmetic expression followed by the token `+` followed by another arithmetic expression. This structure is obviously recursive.

Note that context-free grammars only deal with nonterminals and tokens, they never deal with lexemes. So in the example above the symbols "`+`", "`*`", "`(`", "`)`", "`-`", and "`id`" (for identifiers) are tokens coming from the lexer, not lexemes coming from the source program. In practice we will often use tokens that look very much like lexemes to make it easier for you to understand the context-free grammars, but you should always remember that grammars (and parsers) only deal with nonterminals and tokens, they never deal with lexemes. In this course I will often use upper case letters for nonterminals, and write tokens using a `fixed-size font`.

Writing grammar productions one by one like in the example above is a bit tedious, so we will often use an abbreviated form:

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

and then often also put all the productions for the same nonterminal on a single line:

$$E \rightarrow E \texttt{ + } E \mid E \texttt{ * } E \mid \texttt{(} E \texttt{)} \mid \texttt{-} E \mid \texttt{id}$$

Here the symbol | is used to separate the different possible structures for the nonterminal E. Note that, even though we now have a single line describing the nonterminal E, we still have *five* separate productions! The | symbol is not part of the productions but is just used to separate the productions from one another. The | symbol is therefore neither a nonterminal nor a token, it is simply a symbol we use to write productions (just like the arrow $\rightarrow$).

When a grammar defines multiple nonterminals, then one of these nonterminals has to be designated as the *start symbol*, which is the nonterminal representing the category of all possible programs described by the grammar. Unless otherwise specified, when a grammar defines multiple nonterminals, the start symbol will always be the nonterminal that is specified first (i.e. the nonterminal that appears in the LHS of the first production of the grammar). From the start symbol and by using the different productions in every possible order (by replacing nonterminals with one of their corresponding RHS) we can generate all the possible programs that are syntactically valid. In most cases though this set of programs is infinite in size, since recursive productions correspond to sequences of tokens of any length!

CFGs are powerful enough to describe most programming language constructs. In particular they can describe:

- lists of similar constructs;
- the order in which different constructs must appear;
- nested structures to any depth;
- operator precedence and associativity.

To be more formal, a context-free grammar is a 4-tuple $<N, T, P, S>$, where:

- N is the set of nonterminals, which are grammar symbols used to name particular sets of strings in the language we are defining;
- T is the set of tokens (also called *terminals*), which come from the lexical analysis;

- P is a set of productions, which describe how tokens and nonterminals can be combined to form nonterminals; each production has exactly one nonterminal in its LHS and a sequence of tokens and nonterminals in its RHS;
- S is the start symbol of the grammar, which is a nonterminal used to name the category of whole programs.

Derivations

Given a grammar G, we can generate a string in the language $L(G)$ by using a *derivation*. For example, if we have a grammar like this:

$$E \rightarrow E \texttt{ + } E \mid E \texttt{ * } E \mid \texttt{(} E \texttt{)} \mid \texttt{-} E \mid \texttt{id}$$

then we can write:

	E	start symbol
$\Rightarrow$	E `+` E	using production 1
$\Rightarrow$	`id` `+` E	using production 5
$\Rightarrow$	`id` `+` E `*` E	using production 2
$\Rightarrow$	`id` `+` `id` `*` E	using production 5
$\Rightarrow$	`id` `+` `id` `*` `id`	using production 5

In a derivation, we start with the start symbol of the grammar (E in this example, since it is the only nonterminal) then at each step of the derivation we replace exactly one nonterminal with the sequence of tokens and nonterminals coming from the RHS of some grammar production for the nonterminal we are replacing (we indicate in the rightmost column which grammar production we used at each step). The symbol $\Rightarrow$ means "derive", so for example from the nonterminal E we can use grammar production 1 to derive the sequence of tokens and nonterminals "E `+` E". Each sequence of nonterminals and tokens that we derive at each step of the derivation is called a *sentential form*. The last sentential form in the derivation contains only tokens, and is therefore a *sentence* which is a syntactically correct string in the programming language described by the grammar. We use the $\stackrel{*}{\Rightarrow}$ symbol to mean "derive in zero or more steps". So, for example, $E \stackrel{*}{\Rightarrow}$ `id + id * id`, and `id +` $E \stackrel{*}{\Rightarrow}$ `id + id *` E.

The derivation above is actually a *leftmost derivation*: at every step of the derivation we replaced the leftmost nonterminal in each sentential form to get the next sentential form. In a *rightmost derivation* we replace at every step of the derivation the rightmost nonterminal in each sentential form to get the next sentential form. Here is an example of a rightmost derivation using the same grammar as above, and for the same sentence:

	E	start symbol
$\Rightarrow$	E `+` E	using production 1
$\Rightarrow$	E `+` E `*` E	using production 2
$\Rightarrow$	E `+` E `*` `id`	using production 5
$\Rightarrow$	E `+` `id` `*` `id`	using production 5
$\Rightarrow$	`id` `+` `id` `*` `id`	using production 5

Here are some properties regarding the various kinds of derivations.

- A given sentence can have multiple leftmost derivations if the grammar is ambiguous.
- A given sentence has exactly one leftmost derivation if the grammar is not ambiguous.
- A given sentence can have multiple rightmost derivations if the grammar is ambiguous.
- A given sentence has exactly one rightmost derivation if the grammar is not ambiguous.
- A given sentence can have multiple derivations. There is at least one leftmost derivation and one rightmost derivation (which may or may not be the same) and possibly many other derivations which are neither leftmost nor rightmost.

Since our example grammar for arithmetic expressions above is ambiguous, here is another leftmost derivation of the same sentence using the same grammar:

	E	start symbol
$\Rightarrow$	E `*` E	using production 2
$\Rightarrow$	E `+` E `*` E	using production 1
$\Rightarrow$	`id` `+` E `*` E	using production 5
$\Rightarrow$	`id` `+` `id` `*` E	using production 5
$\Rightarrow$	`id` `+` `id` `*` `id`	using production 5

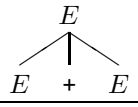
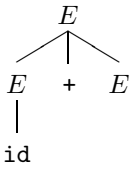
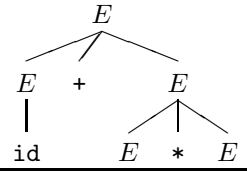
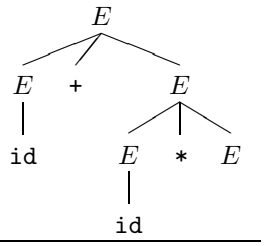
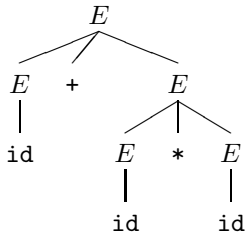
And here is a derivation for the same sentence which is neither leftmost nor rightmost:

	E	start symbol
$\Rightarrow$	E `+` E	using production 1
$\Rightarrow$	E `+` E `*` E	using production 2
$\Rightarrow$	E `+` `id` `*` E	using production 5
$\Rightarrow$	`id` `+` `id` `*` E	using production 5
$\Rightarrow$	`id` `+` `id` `*` `id`	using production 5

In this derivation we replaced the rightmost E to go from the second sentential form to the third one, replaced the middle E to go from the third sentential form to the fourth one, and replaced the leftmost E to go from the fourth sentential form to the fifth.

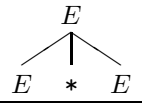
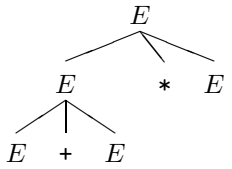
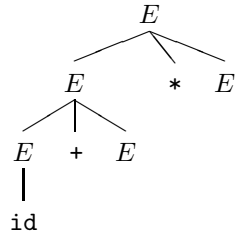
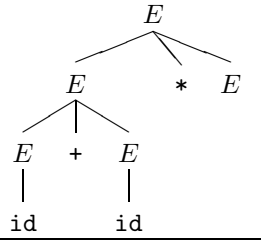
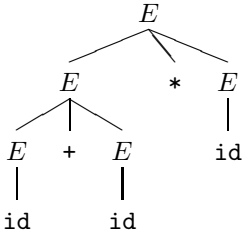
Parse Trees and Ambiguity

From a given derivation of a sentence, we can construct a corresponding *parse tree*, which describes the structure of the sentence. Each step in the derivation then corresponds to extending exactly one nonterminal in the parse tree with one or more new subtrees. For example, using the first leftmost derivation above:

	E	start symbol	E
$\Rightarrow$	E `+` E	using production 1	
$\Rightarrow$	`id` + E	using production 5	
$\Rightarrow$	`id` + E * E	using production 2	
$\Rightarrow$	`id` + `id` * E	using production 5	
$\Rightarrow$	`id` + `id` * `id`	using production 5	

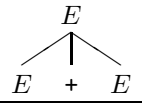
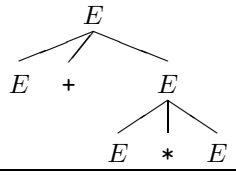
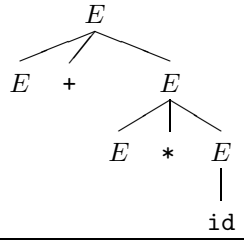
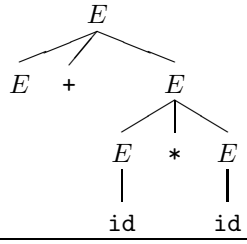
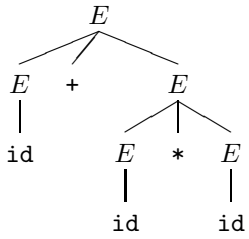
When constructing a parse tree like this, always make sure that the tokens that form the sentence you derived all appear at the leaves of the parse tree, from left to right, and that all the interior nodes in the tree are nonterminals. If, when reading the leaves of the parse tree from left to right, some tokens are missing, or there are too many tokens, or some nonterminal is a leaf, then you know you made a mistake.

As we have seen above, our grammar for arithmetic expressions is ambiguous, so here is the parse tree corresponding to the second leftmost derivation we wrote above:

	E	start symbol	E
$\Rightarrow$	E `*` E	using production 2	
$\Rightarrow$	E `+` E `*` E	using production 1	
$\Rightarrow$	`id` `+` E `*` E	using production 5	
$\Rightarrow$	`id` `+` `id` `*` E	using production 5	
$\Rightarrow$	`id` `+` `id` `*` `id`	using production 5	

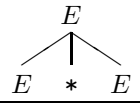
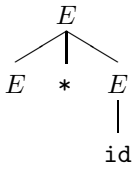
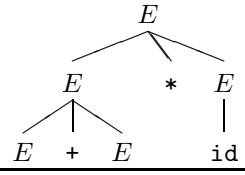
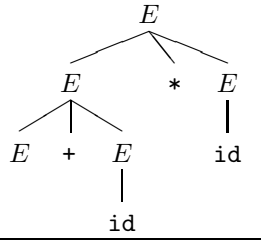
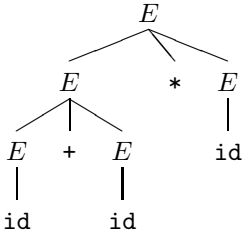
As you can see, we have here two different (leftmost) derivations that correspond to two different parse trees for the same sentence. This is a proof that the grammar we use is ambiguous. By definition, a grammar is *ambiguous* if there is some sentence in the language described by the grammar for which it is possible to construct more than one possible parse tree (using any kind of derivation). Obviously, since constructing a parse tree for the source program is the goal of the syntax analysis in a compiler, we want to avoid using an ambiguous grammar in our syntax analysis...

If we construct the parse tree corresponding to the rightmost derivation above:

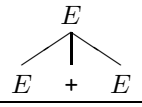
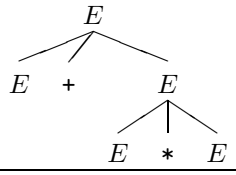
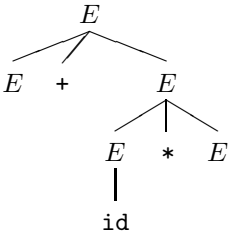
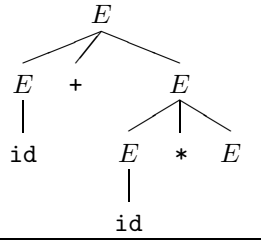
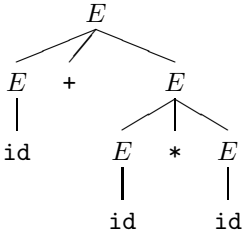
	E	start symbol	E
$\Rightarrow$	E `+` E	using production 1	
$\Rightarrow$	E `+` E `*` E	using production 2	
$\Rightarrow$	E `+` E `*` `id`	using production 5	
$\Rightarrow$	E `+` `id` `*` `id`	using production 5	
$\Rightarrow$	`id` `+` `id` `*` `id`	using production 5	

then we get the same parse tree as the first leftmost derivation above (which is not a coincidence, as each parse tree always corresponds to one leftmost and one rightmost derivation). What changes is the order in which the different parts of the parse tree are constructed.

Since the grammar is ambiguous, here is another rightmost derivation (which we have not seen yet) that constructs another parse tree, which turns out (again, not by coincidence) to be the same parse tree constructed by the second leftmost derivation above:

	E	start symbol	E
$\Rightarrow$	E `*` E	using production 2	
$\Rightarrow$	E `*` `id`	using production 5	
$\Rightarrow$	E `+` E `*` `id`	using production 1	
$\Rightarrow$	E `+` `id` `*` `id`	using production 5	
$\Rightarrow$	`id` `+` `id` `*` `id`	using production 5	

Finally, for completeness, here is the parse that is constructed from the last derivation in the previous section, which is neither leftmost nor rightmost:

	E	start symbol	E
$\Rightarrow$	E `+` E	using production 1	
$\Rightarrow$	E `+` E `*` E	using production 2	
$\Rightarrow$	E `+` `id` `*` E	using production 5	
$\Rightarrow$	`id` `+` `id` `*` E	using production 5	
$\Rightarrow$	`id` `+` `id` `*` `id`	using production 5	

This derivation constructs the same parse as the first leftmost derivation above, but this is really a coincidence.

Here are some properties regarding parse trees and the various kinds of derivations.

- Each derivation (leftmost, rightmost, or otherwise) corresponds to exactly one parse tree, whether the grammar is ambiguous or not. This is obvious since each step of a derivation tells us exactly what should be added to the corresponding parse tree and where.
- Each parse tree corresponds to multiple derivations, whether the grammar is ambiguous or not.

- Each parse tree corresponds to exactly one leftmost derivation and exactly one rightmost derivation, whether the grammar is ambiguous or not. The leftmost derivation can be reconstructed (backwards) from the parse tree by doing a bottom-up right-to-left traversal of the parse tree. The rightmost derivation can be reconstructed (backwards) from the parse tree by doing a bottom-up left-to-right traversal of the parse tree (we will see that this is in fact what a bottom-up parser does).

- All derivations of the same sentence correspond to the same parse tree if the grammar is not ambiguous.

- Multiple derivations of the same sentence may or may not correspond to the same parse tree if the grammar is ambiguous.

So, if a grammar is unambiguous, then a given sentence has a unique leftmost derivation, a unique rightmost derivation, and both derivations correspond to the same unique parse tree (though the different parts of the unique parse tree might be constructed in a different order depending on whether the leftmost or rightmost derivation is used to construct the parse tree). The parse tree corresponds to many derivations but has a unique leftmost derivation and a unique rightmost derivation.

If a grammar is ambiguous, then there might be multiple leftmost derivations, multiple rightmost derivations, and many parse trees for the same sentence. Different leftmost derivations must correspond to different parse trees. Different rightmost derivations must correspond to different parse trees. Leftmost and rightmost derivations may or may not correspond to the same parse tree. Each parse tree corresponds to many derivations but still only has one leftmost derivation and only one rightmost derivation (which is why different leftmost derivations must correspond to different parse trees, and similarly for rightmost derivations).

Be careful: the previous paragraph means that a leftmost derivation of a sentence and a rightmost derivation of a sentence can correspond to the same parse tree, but that does not prove that the grammar is not ambiguous (as we have seen with our examples above). It just means that you were lucky (or unlucky) in using a rightmost derivation that just happened to correspond to the same parse tree as the leftmost derivation you used. To prove that a grammar is unambiguous, you need to prove that each sentence has a unique leftmost derivation and unique rightmost derivation (and therefore a unique parse tree), and you need to prove this for *all* sentences in the programming language... In general deciding whether a grammar is ambiguous or not is undecidable (i.e. there is no algorithm which, when given a grammar as input, can always tell you whether the grammar is ambiguous or not). Fortunately, the algorithms we are going to use to create parsers will be able to tell us when they can prove that a grammar is unambiguous which will be good enough for us (if the algorithms cannot prove that a grammar is unambiguous, then it will mean that either the grammar is ambiguous or that the algorithms are not powerful enough).

If a grammar turns out to be ambiguous, then we will have two choices:

- modify the grammar by hand to transform it into an unambiguous grammar and then generate an unambiguous parser from that; this is doable (we will see how to do this for simple cases related to the precedence and associativity of operators, for example) but it is easy to make mistakes when rewriting a grammar and the result is often complicated to understand;

- keep the ambiguous grammar and generate an ambiguous parser from it, then transform the ambiguous parser into an unambiguous one either manually or by giving some extra information about what we want to the tool used to generate the parser; we will see that transforming the parser is often easier to do than the previous solution, and tools like YACC or Bison in fact provide special keywords that you can use in your `.y` files to transform an ambiguous parser into an unambiguous one.

We have seen that our simple grammar for arithmetic expressions above is ambiguous. Here is another well known example of an ambiguous construct:

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

where S represents a statement, E represents an expression, and the token `other` represents other kinds of statements (which we do not care to specify explicitly here since they are mostly irrelevant to this example).

This grammar for `if` statements is simple to understand but it is ambiguous, because a sentence like the following one:

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

can be parsed in two different ways, depending on whether the `else` is associated with the first or the second `if`. This problem can be solved by rewriting the grammar:

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

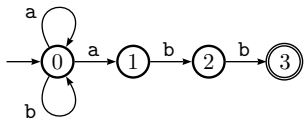
where M represents a "matched" statement where every `if` has a matching `else`, and U represents an "unmatched" statement where some `if` does not have a matching `else`. This new grammar removes the possibility of having statements of the form "`if ( E )` U `else` M" or "`if ( E )` U `else` U" and ensures that there is always an M before the `else`. This in turn ensures that the `else` always has to be associated with the `if`, since the M cannot contain any `if` which does not already have its own associated `else`.

This new grammar is not so easy to understand though, and it takes some thinking to be convinced that it works and is unambiguous. Another simpler

way to solve the same problem is therefore to keep the original ambiguous grammar and then simply provide the tool used to generate the corresponding parser with some extra information that indicates that `else` should always be associated with the closest `if` (this can be done by giving a high precedence to the `else` token...)

Limits of Context-Free Grammars

There is a theorem in automata theory that says that for every regular expression there is an equivalent context-free grammar. For example, we know from the previous lecture notes that the regular expression (`a`|`b`)*`abb` can be implemented using the following NFA:



This NFA can then be simulated using the following CFG:

$$
\begin{array}{lcl}
S_0 & \rightarrow & \texttt{a}\ S_0 \mid \texttt{b}\ S_0 \mid \texttt{a}\ S_1 \\
S_1 & \rightarrow & \texttt{b}\ S_2 \\
S_2 & \rightarrow & \texttt{b}\ S_3 \\
S_3 & \rightarrow & \varepsilon
\end{array}
$$

where each state of the NFA is represented by one nonterminal in the CFG, and each transition of the NFA is represented by one production in the CFG.

This means that in theory everything done in the lexical analysis phase of a compiler could instead be done as part of the syntax analysis phase. As we have said in the previous lecture notes though, in practice both phases are kept separate because:

- Separating the two makes each simpler to implement. Implementing a single phase that combined both lexing and parsing at the same time would make that piece of software very complex. For example, trying to recognize the `else` keyword using a CFG would require five new nonterminals in the grammar. Writing a CFG that took into account all the regular-expressions we have seen for identifiers, keywords, integers, operators, etc., in addition to taking care of possible spaces, newlines, and comments in the input, would soon result in a CFG with thousands of nonterminals and productions. This puts a practical limit (but no theoretical limit) on merging lexers and parsers.

- If the lexer is kept separate from the parser, then it is easier to change the lexer to do a lot of input buffering to increase the speed of the compiler, or to deal with special encodings for the source program. There is then no need to worry about these things in the parser, since the parser is then isolated from the way the lexemes in the source code are actually written.

As we have said above, there are other types of grammars which are more powerful than CFGs. For example there exists no context-free grammar that can describe the set of strings {`wcw`, where `w` is a string in (`a`|`b`)*}. In other words, a CFG cannot be used to check whether some string `w` from some set (`a`|`b`)* of strings appears twice in your program or not. If you think of (`a`|`b`)* as representing the set of all possible variable names in a programming language, and of `w` as representing the name of some specific variable in your source program, then this means a CFG cannot be used to check whether the same variable name appears twice in your program. This in turn explains why CFGs cannot be used to check that a variable is declared before it is used in programming languages like C or Java. Checking that a variable is declared before it is used implies being able to check that the variable name appears at least twice in the program (once for the declaration, once for the use) which, as we just saw, is not something a CFG can do. That is why such a check has to be done in the semantic analysis phase of a compiler, it cannot be done in the syntax analysis phase.

There are other kinds of more complex grammars that can be used to describe complex sets of strings like the set {`wcw`, where `w` is a string in (`a`|`b`)*} we saw above. Unfortunately creating parsers for such grammars is either very difficult or impossible, and even when it is possible those parsers are usually very inefficient. So in practice the syntax of most programming languages is based on context-free grammars. As we have said above, in most cases CFGs are good enough to describe the kind of programming language constructs we are interested in, so we do not lose much by restricting ourselves to CFGs.

Top-Down Parsing

Now that we understand how context-free grammars work, we can start looking at how to implement parsers for them. As we have said in the introduction, we are going to look first at top-down parsers, because top-down parsers are easy to write by hand, at least for a restricted set of CFGs called the LL(1) grammars.

In this section we are in fact going to look at three different kinds of top-down parsers. First we will look at recursive-descent parsers, then we will look at a restricted version of recursive-descent parsers called predictive parsers which are recursive too but do not require backtracking. Then we will look at nonrecursive predictive parsers, which can implement the same grammars as (recursive) predictive parsers but without using recursion (as their name indicates...)

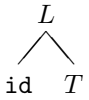
Before we do this though, let's first look quickly at how top-down parsers work in general. Here is a grammar for a list of identifiers separated by commas (a list of variables in an integer variable declaration in Java, for example), where L represents a complete list of identifiers, T represents the "tail" of such a list (a list of identifiers that starts with a comma instead of starting with an identifier), and `id` is a token representing a single identifier:

$$
\begin{array}{lcl}
L & \rightarrow & \texttt{id}\ T \\
T & \rightarrow & \texttt{,}\ \texttt{id}\ T \\
 & | & \texttt{;}
\end{array}
$$

Supposed the input string to parse is "`id , id , id ;`". A top-down parser will start with the start symbol in the grammar which is L in our case:

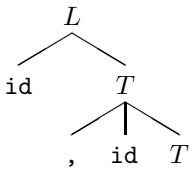
$$L$$

Next the top-down parser will try to guess which production to use to replace L with something more precise, so as to build the parse tree. There is only one production that has L as its LHS, so the parser replaces L with:



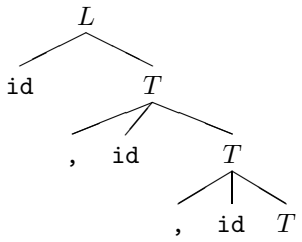
This means the first token in the input must be an `id` literal. Fortunately the first token in the input is `id` so everything is fine and that first token is consumed (i.e. it disappears from the input stream of tokens). Otherwise the parser would have generated a syntax error saying that no identifier was found where one was expected.

Next the top-down parser has to find a production to replace T. There are two productions that have T as their LHS, so the parser has to peek at the next token in the input (without consuming it) to try to guess which one of the two productions to use. The next token in the input stream is a comma, so the parser knows it has to use the second production for T in the CFG:

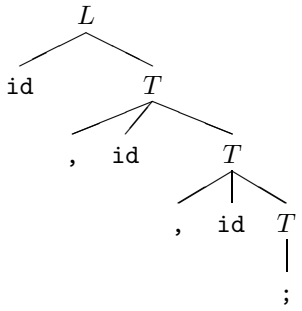


Peeking at the next input token like this (without consuming it) to try to guess which production to use next is called doing some *lookahead*. Parsers often use lookahead to decide what to do next.

Once the parse tree has been extended, the parser first checks that the next token in the input is a comma, because that is what is expected from the current parse tree. Obviously this is the case, since the parser had already peeked at that next token in the previous step to decide which grammar production to use. So the comma from the input can be consumed. Next, the parser knows from the current parse tree that there should be another identifier following the comma, followed by another T. The parser checks that the next input token really is an identifier (and in this example it is fine since the next token is `id`, so `id` is now consumed) and the parser next has again to find which production to use to expand the parse tree under the T. Again the top-down parser peeks at the next input token, which is again a comma, uses the second production for T in the CFG, and the parse tree becomes:



At that point the parser again has to peek at the next token in the input to determine which of the two productions for T to use. This time the lookahead token is `;`. The parser then knows the next production to use is the last one in the grammar, and the parse tree becomes:



The parser can then check the token `;` in the parse tree against the next token in the input, and again this check obviously succeeds, since the next token in the input was used to decide how to extend the last T in the parse tree. So the parser can consume the last input token `;`.

At that point no nonterminals remain in the leaves of the parse tree, all tokens in the parse tree have been checked against token in the input, and no tokens are left in the input (they have all been consumed). The parsing is therefore over, and the current parse tree is the final parse tree for the input sentence.

In essence a top-down parser reads the input from left to right and tries to construct a parse tree by finding a *leftmost derivation* of the input: the leftmost nonterminal in the current parse tree is always the one the parser tries first to extend further towards the bottom of the tree. The parser starts from the start symbol (the "top" of the grammar) and tries to work its way down (as can be expected from a "top-down" parser) by using a succession of guesses based on peeking at the next token in the input (the lookahead token), to try to guess which production to use next to extend the parse tree towards the leaves.

The grammar in this example is in fact what is called an LL(1) grammar, because the top-down parser for it reads input from "Left to right" and computes a "Leftmost derivation" using only *one* token of lookahead.

Note that the grammar we used in our example above:

$$
\begin{array}{lcl}
L & \rightarrow & \texttt{id}\ T \\
T & \rightarrow & \texttt{,}\ \texttt{id}\ T \\
 & | & \texttt{;}
\end{array}
$$

could be made simpler:

$$
\begin{array}{lcl}
L & \rightarrow & \texttt{id}\ \texttt{;} \\
 & | & \texttt{id}\ \texttt{,}\ L
\end{array}
$$

The reason for using the first grammar for top-down parsing rather than the second one is that in the first grammar all the RHS of the different productions start with different tokens ("`id`", "`,`", and "`;`"). This means the top-down parser can decide which production to use next by using only one token of lookahead. In the second grammar both productions start with the same token "`id`" which means the parser would need a second token of lookahead after that ("`,`" or "`;`") to be able to decide which of the two productions to use next. Using the first grammar to automatically generate a top-down parser therefore leads to a more efficient parser (fortunately there is a way to automatically transform the second grammar into the first one, by finding common prefixes of the RHS of productions and inserting new nonterminals to represent those prefixes, as we are going to see below).

Recursive-Descent Parsers

A recursive-descent parser is a simple generalization of the kind of top-down parser we have just seen in the previous example. The only difference is that, when using a recursive-descent parser, some *backtracking* might be necessary.

Let's look at an example. Consider the following grammar:

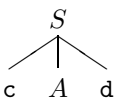
$$
\begin{array}{lcl}
S & \rightarrow & \texttt{c}\ A\ \texttt{d} \\
A & \rightarrow & \texttt{a}\ \texttt{b} \mid \texttt{a}
\end{array}
$$

where `a`, `b`, `c`, and `d` are tokens. Using this grammar we can now try to parse the input `cad` using the same method as in our previous example.

We start with the start symbol of the grammar S:

$$S$$

The top-down parser then can only use the first grammar production to extend the parse tree:

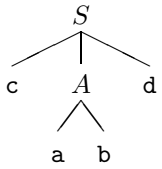


Next the parser matches the token `c` in the parse tree with the first token `c` in the input, and therefore consumes this token.

The top-down parser now has to figure out how to extend the parse tree under the nonterminal A. To do this the parser has to decide which one of the

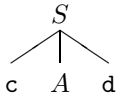
two productions for A to use next. So the parser peeks at the next token in the input (the lookahead token), which is `a`, and tries to decide which of the two productions for A to use based on that information. Unfortunately, both productions for A start with the token `a`, so at the point the top-down parser has no other choice but to *guess* which production to use.

Suppose the parser (wrongly) decides to use the first production for A. Then the parse tree becomes:



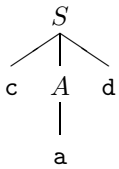
The parser then checks that the token `a` in the parse tree matches the next token in the input. This obviously works, since the next token in the input is precisely the lookahead token `a` that the parser used to try to make its decision. So the parser consumes this token `a` from the input and moves on the next token. The next token to check in the parse tree is `b`. Unfortunately the next token in the input is `d`, which does not match the expected token `b` from the parse tree.

At this point the top-down parser has discovered that it made the wrong choice when it had to decide which production to use for A. So the parser now has to backtrack, which means it has to undo all the work it did since the point in time when it made the wrong decision. This means the parser has to move the parser tree back to its previous form:



and the parser also has to put back into the input the token `a` that it had just consumed.

Once it has backtracked and undone all this work, the parser can then try the other production for A. The parse tree then becomes:



The parser then checks again that the token `a` in the parse tree matches the next token in the input, and again this check obviously succeeds since the next token in the input was used as the lookahead token (even though it did not provide the parser with enough information to make the right decision). So the parser consumes the input token `a` (again, after having consumed that token a first time and then having put it back into the input when backtracking).

After that the top-down parser still has to check that the token `d` in the parse tree matches the next token in the input. Again this is the case, so the last input token `d` is consumed. At this point there are no nonterminals in the parse tree that are left to extend, all tokens in the parse have been checked against tokens in the input, and all tokens in the input have been consumed. The parser therefore accepts the input string, and the parse tree it has constructed correctly describes the structure of the input string.

As we can see in this example, top-down recursive-descent parsers work just like in our previous example in our introduction to top-down parsing above. In the general case a recursive-descent parser might need to backtrack and undo some of its work though, when looking at the lookahead token is not enough for the parser to decide exactly which grammar production to use next. If backtracking is needed, then obviously the corresponding grammar is not LL(1): the parser reads input from left-to-right (L) and finds a leftmost derivation (L) of the input, but a single token of lookahead (1) is not enough to always decide which production the parser should use at every step. In our example just above, the grammar is in fact LL(2) since two tokens of lookahead are enough to always decide which production to use for the nonterminal A: if the next two tokens in the input are `ab` then the first production for A should be used, if the next two tokens in the input are `ad` then the second production for A should be used, and all the other cases are a syntax error.

Implementation of Recursive-Descent Parsers

Recursive-descent parsers are called that way because they can be fairly easily implemented using recursive functions. The idea is to create one function for each nonterminal in the grammar. If a nonterminal has a single production (as is the case for S in the example above) then the function for this nonterminal directly tries to match the input with the RHS of that production: tokens in the RHS are directly matched one at a time with tokens in the input, and nonterminals in the RHS are matched by calling the function for that nonterminal. If a nonterminal has several productions then the function for this nonterminal uses the lookahead token (i.e. peeks at the next token in the input without consuming it) to decide which production to use. Once a production has been chosen, it is used by the function to match input as in the previous case. If the lookahead token is not enough for the function to be able to decide which production to use, then some backtracking has to be implemented (using `try-catch` in the example below, but that is not the only possible way, or the simplest one).

In essence each function call corresponds to the parser expecting the next tokens in the input to have the structure described by the nonterminal associated with the function being called. A function return then corresponds to the parser having just finished matching a sequence of input tokens that has the structure described by the nonterminal associated with the function that is returning. As such, the stack of function calls always describes what the parser expects to see at various points in the remaining input. In particular the current function being called indicates what the parser expects to see next in the input.

Of course, as the parser calls the various functions for the different nonterminals, these functions also have to build the parse tree. So each function in the code should create a new parse tree node. If the node is for a token (i.e. the node is a leaf of the parse tree) then the node can simply contain the token itself. If the node is for a nonterminal, then the node should contain pointers to all the nodes for all the tokens and nonterminals in the RHS of the grammar production used by the function to match the input.

For example, for the grammar for the previous example:

$$\begin{array}{lcl} S & \rightarrow & \texttt{c}\ A\ \texttt{d} \\ A & \rightarrow & \texttt{a}\ \texttt{b} \mid \texttt{a} \end{array}$$

one creates the following code:

```
match_token(expected_token){
  next_input_token = consume_next_input_token();
  if(expected_token == next_input_token){
    return new_token_node(expected_token);
  }else{
    raise_syntax_error();
  }
}

match_S(){
  /* only one production so no need for lookahead token */
  c_node = match_token('c');
  A_node = match_A();
  d_node = match_token('d');
  return new_S_node(c_node, A_node, d_node);
}

match_A(){
  lookahead_token = peek_at_next_input_token();
  if(lookahead_token == 'a'){
    try{
      a_node = match_token('a'); /* always works */
      b_node = match_token('b'); /* may raise a syntax error */
      return new_A_node(a_node, b_node);
    }catch(syntax_error){
      retract_some_input_tokens(2);
      a_node = match_token('a'); /* always works */
      return new_A_node(a_node);
    }
  }else{
    raise_syntax_error();
  }
}
```

```
parser_main(){
  parse_tree = match_S(); /* S is the start symbol */
}
```

In this example, the parser calls the `match_S` function, since S is the start symbol of the grammar. That function creates a new parse tree node for an S nonterminal. This parse tree node contains three pointers to three nodes that are the roots of three different subtrees, one for `c` (constructed by the `match_token` function on a correct match), one for A (constructed by the `match_A` function on a correct match), and one for `d` (constructed again by the `match_token` function on a correct match). Similarly for the `match_A` and `match_token` functions, which both construct new nodes on successful matches.

The problem with this code is that implementing the backtracking can quickly become very painful: the code has to precisely know how many input tokens it might have to put back into the input when backtracking, and then the code always has to remember at least that many input tokens when consuming them just in case it later has to put them back into the input. For example, in the code for `match_A` above, two tokens might have to be put back into the input using the `retract_some_input_tokens` function. This means that the `consume_next_input_token` function always has to store somewhere (in some global variable) the last two tokens it has consumed, so that later the `retract_some_input_tokens` function knows which tokens to put back into the input when necessary.

This problem of properly implementing backtracking quickly becomes worse if many nonterminals require backtracking (in which case many functions will need to use `try-catch`) or if a nonterminal requiring backtracking has many productions that have to be tried one after the other (in which case the function for that nonterminal will have to use many nested `try-catch` statements). An additional problem with backtracking is that it makes the parser inefficient: for example in the code above the input token `a` might get matched in the `try` branch of the `match_A` function, then it might be put back into the input and matched again in the `catch` branch of the function. So backtracking requires undoing some work to backtrack, then re-doing some work when trying another production.

The result is that in practice people seldom use recursive-descent parsers, simply because backtracking is not something people want to have to deal with. Instead what people use is a restricted version of recursive-descent parsers called predictive parsers. As the name suggests, predictive parser can always predict which production to use at every step of the parsing process (after looking at most at one lookahead token) without ever needing to backtrack. So before we look at how predictive parsers work, we want to understand first how to write a grammar that does not require the corresponding top-down parser to backtrack.

Left-Factoring

If we look again at the grammar from the previous example:

$$\begin{array}{lcl} S & \rightarrow & \texttt{c}\ A\ \texttt{d} \\ A & \rightarrow & \texttt{a}\ \texttt{b} \mid \texttt{a} \end{array}$$

we see that the corresponding top-down recursive-descent parser needs to backtrack because the two productions for A both start with the token `a`. Since they both start with the same token, peeking at the lookahead token (the next token in the input) does not give enough information to the parser to be able to decide which of the two productions to use next. To solve this problem, we want to transform the grammar so that there is only one production for A that starts with the token `a`. To do this, we factor the common prefix between the two productions, and then introduce a new nonterminal A' that takes care of the remaining suffixes:

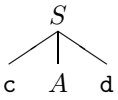
$$\begin{array}{lcl} S & \rightarrow & \texttt{c}\ A\ \texttt{d} \\ A & \rightarrow & \texttt{a}\ A' \\ A' & \rightarrow & \texttt{b} \mid \varepsilon \end{array}$$

Then the parser only has to make a decision between different productions once it reaches the new nonterminal A'. This in essence means that the decision the parser has to make is now delayed by one input token, which is just enough for the parser to then make the right decision. This grammar transformation is called *left factoring*.

Given the input string `cad` and this left factored grammar, the parser then starts as before with the start symbol of the grammar S:

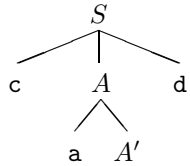
$$S$$

The top-down parser then can only use the first grammar production to extend the parse tree:

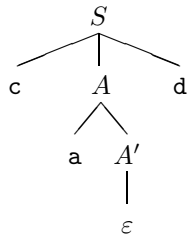


Next the parser matches the token `c` in the parse tree with the first token `c` in the input, and therefore consumes this token.

The top-down parser now has to figure out how to extend the parse tree under the nonterminal A. There is now only one possible choice with the new left factored grammar, which is to use the second production in the grammar:



The parser then checks the token `a` in the parse tree against the token `a` in the input and everything is fine so the input token `a` is consumed. Next the parser has to extend the parse tree under the A' nonterminal. Since there are two possible productions for A', the parser looks at the lookahead token, which is the input token `d`. Since the first grammar production for A' would require the lookahead token to be a `b`, the parser knows it has to use the other production for A':



Then the parser only has to check that the token `d` in the parse tree matches the next input token `d`, which is obviously the case. The input has then been completely parsed without any need for backtracking. In essence the choice that the parser had to make was delayed until the parser reached A', which was *after* the parser matched the token `a` (which used to be the token that required the parser to make a choice).

Since backtracking is no longer required, the corresponding code becomes simpler and directly follows the structure of the grammar:

```
match_token(expected_token){
  next_input_token = consume_next_input_token();
  if(expected_token == next_input_token){
    return new_token_node(expected_token);
  }else{
    raise_syntax_error();
  }
}

match_epsilon(){
  return new_epsilon_node(); /* epsilon always matches */
}

match_S(){
  /* only one production so no need for lookahead token */
  c_node = match_token('c');
  A_node = match_A();
  d_node = match_token('d');
  return new_S_node(c_node, A_node, d_node);
}

match_A(){
```

```
  /* only one production so no need for lookahead token */
  a_node = match_token('a');
  Ap_node = match_Ap();
  return new_A_node(a_node, Ap_node);
}

match_Ap(){
  /* two possible productions so use lookahead token to decide */
  lookahead_token = peek_at_next_input_token();
  if(lookahead_token == 'b'){
    b_node = match_token('b');
    return new_Ap_node(b_node);
  }else{
    e_node = match_epsilon();
    return new_Ap_node(e_node);
  }
}

parser_main(){
  parse_tree = match_S(); /* S is the start symbol */
}
```

More formally, *left factoring* is defined as follows. For each nonterminal A which has multiple productions starting with the same prefix α:

$$A \quad \rightarrow \quad \alpha\ \beta_1 \mid \ldots \mid \alpha\ \beta_k \mid \gamma_1 \mid \ldots \mid \gamma_n$$

where α is a non-empty sequence of grammar symbols (tokens and nonterminals), and $\beta_1, \ldots, \beta_k$ and $\gamma_1, \ldots, \gamma_n$ are possibly empty sequences of grammar symbols; create a new nonterminal A' and transform the grammar as follows:

$$\begin{array}{lcl} A & \rightarrow & \alpha\ A' \mid \gamma_1 \mid \ldots \mid \gamma_n \\ A' & \rightarrow & \beta_1 \mid \ldots \mid \beta_k \end{array}$$

Then repeat the process until no nonterminal has two or more productions starting with the same prefix.

As another example, if we look again at the simple version of the grammar for lists of identifiers that we talked about above in the introduction to top-down parsing:

$$\begin{array}{lcl} L & \rightarrow & \texttt{id ;} \\ & \mid & \texttt{id ,}\ L \end{array}$$

we can see that L has two productions starting with `id`. Left factoring this grammar then gives us:

$$\begin{array}{lcl} L & \rightarrow & \texttt{id}\ L' \\ L' & \rightarrow & \texttt{;} \\ & \mid & \texttt{,}\ L \end{array}$$

We can see that this new grammar is almost identical to the slightly more complex grammar we actually used when parsing the example input "`id , id , id ;`":

$$\begin{array}{lcl} L & \rightarrow & \texttt{id}\ T \\ T & \rightarrow & \texttt{,}\ \texttt{id}\ T \\ & | & \texttt{;} \end{array}$$

The only difference is the name of the second nonterminal (T instead of L') and the fact that L in the third production of the former grammar has been replaced by its RHS "`id` T" in the second production of the latter grammar.

Left Recursion Elimination

Before we look at predictive parsers, there is another more subtle problem that we need to handle with regard to grammars. Consider for example the following grammar:

$$A \rightarrow A\ \texttt{a} \mid \texttt{b}$$

The two productions in this grammar do not have any common prefix, so there is no need to do any left factoring. If we then write the function corresponding to the nonterminal A we get the following code:

```
match_A(){
  /* two possible productions so use lookahead token to decide */
  lookahead_token = peek_at_next_input_token();
  if(lookahead_token == '???'){
    A_node = match_A();
    a_node = match_token('a');
    return new_A_node(A_node, a_node);
  }else{
    b_node = match_token('b');
    return new_A_node(b_node);
  }
}
```

The problem then is: what do we replace the `???` with in the test of the lookahead token? Based on the grammar, we want the parser to be able to recognize the set of input strings {`b`, `ba`, `baa`, `baaa`, ...}. If we replace the `???` in the code with `a` then, when given input like `baa`, the lookahead token will be `b`, the test will be false, only the first token `b` in the input will be matched in the `else` branch of the code, the remaining input tokens `aa` will not be matched, and the input string will therefore be rejected by the parser. In fact, among all the strings in {`b`, `ba`, `baa`, `baaa`, ...} that we want the parser to accept, only the first one will be indeed accepted, which is clearly not satisfactory.

The only other solution is to replace the `???` with `b`, but then we have another problem: if we give again to the parser the input `baa`, the lookahead

token will be `b`, the test will be true, and the parser will then immediately call again the `match_A` function without consuming any input token. Since no input token will have been consumed, the lookahead in the recursive call to `match_A` will still be `b`, the test will therefore still be true, and `match_A` will just calls itself recursively a second time. In fact `match_A` will just keep calling itself for ever (or at least until the parser runs out of stack space) since no input will ever be consumed and the lookahead token therefore will never change to stop the recursion.

This example shows that recursive-descent parsers (and in fact all top-down parsers) cannot deal with grammars of the form above, even when the grammar is left factored. The grammar above is left recursive, which means that a production for some nonterminal in the grammar has a RHS that starts with the same nonterminal. It is not possible to use top-down parsers with left recursive grammars, and therefore such grammars have to be transformed to eliminate left recursion before they can be used to create a top-down parser.

More formally, a grammar is *left recursive* if there is some nonterminal A and some productions in the grammar such that $A \stackrel{*}{\Rightarrow} A\alpha$, where α is some sequence of grammar symbols.

In the example above, the grammar is *immediately left recursive* because we directly have $A \rightarrow A$ `a`, but left recursion can be more complicated. For example the following grammar is left recursive:

$$\begin{array}{lcl} S & \rightarrow & A\ \texttt{a} \mid \texttt{b} \\ A & \rightarrow & \texttt{c} \mid S\ \texttt{d} \end{array}$$

because $S \Rightarrow A$ `a` $\Rightarrow S$ `d a`, so $S \stackrel{*}{\Rightarrow} S$ `d a`.

There are algorithms to eliminate left recursion in this more general case, but in these lecture notes we are only going to look at how to eliminate left recursion in the immediate case.

In the immediate case, *left recursion elimination* is defined as follows. For each nonterminal A which has one or more productions with RHSs starting with the same nonterminal A:

$$A \quad \rightarrow \quad A\ \alpha_1 \mid \ldots \mid A\ \alpha_k \mid \beta_1 \mid \ldots \mid \beta_n$$

where $\alpha_1, \ldots, \alpha_k$ and $\beta_1, \ldots, \beta_n$ are possibly empty sequences of grammar symbols; create a new nonterminal A' and transform the grammar as follows:

$$\begin{array}{lcl} A & \rightarrow & \beta_1\ A' \mid \ldots \mid \beta_n\ A' \\ A' & \rightarrow & \alpha_1\ A' \mid \ldots \mid \alpha_k\ A' \mid \varepsilon \end{array}$$

Then repeat the process until no nonterminals are left recursive anymore. Note: do not forget the ε production when transforming the grammar!

In the transformed grammar the nonterminal A still represents the same set of strings as before but the nonterminal is no longer left recursive. The new nonterminal A' is right recursive but this is not a problem for top-down parsers.

So, going back to our example grammar:

$$A \quad \rightarrow \quad A\ \texttt{a} \mid \texttt{b}$$

Eliminating left recursion from this grammar gives us the following new grammar:

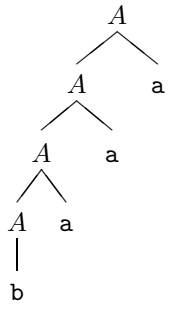
$$\begin{array}{lcl} A & \rightarrow & \texttt{b}\ A' \\ A' & \rightarrow & \texttt{a}\ A' \mid \varepsilon \end{array}$$

This new grammar then corresponds to the following code:

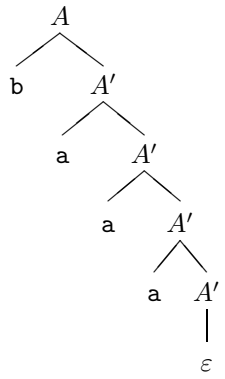
```
match_A(){
  /* only one production so no need for lookahead token */
  b_node = match_token('b');
  Ap_node = match_Ap();
  return new_A_node(b_node, Ap_node);
}

match_Ap(){
  /* two possible productions so use lookahead token to decide */
  lookahead_token = peek_at_next_input_token();
  if(lookahead_token == 'a'){
    a_node = match_token('a');
    Ap_node = match_Ap();
    return new_Ap_node(a_node, Ap_node);
  }else{
    e_node = match_epsilon();
    return new_Ap_node(e_node);
  }
}
```

This code then never goes into an infinite recursive loop, because it always uses the `match_token` function to consume an input token before doing another call to one of the `match_A` or `match_Ap` functions.

From the point of view of parse trees, eliminating left recursion corresponds to transforming the following parse tree (for the input sentence `baaa`):



which is impossible for a top-down parser to generate, into the following parse tree:



which top-down parsers can easily generate.

As another example, consider the following grammar for arithmetic expressions, where * has higher precedence than +, and both + and * are left associative:

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

This grammar is left factored but it is also left recursive, both for the E and T nonterminals. Using the rewriting rule above on E, we transform this grammar into:

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

and then using the same rewriting rule on T this time we get:

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

which then is no longer left recursive.

Left Recursion Elimination and Operator Associativity (Optional)

Note that by doing this, the associativity of + and * has changed: the two operators are now right associative! For operators like + or * this might not be too much of a problem, but for operators like - or / we have to be much more careful, because changing the associativity of these operators might change the meaning of the program... If we need to preserve the left associativity of some operator that corresponds to some left recursive nonterminal (like + corresponds

to E and `*` corresponds to T in the above example) then we cannot eliminate left recursion for these nonterminals, and therefore we cannot use pure top-down parsers with such grammar. Our only solutions are then either to use a bottom-up parser (bottom-up parsers do not care about left recursion or right recursion in grammars) or implement a parser which is almost top-down except that it does not do recursive function calls for the nonterminals for which we need to preserve left associativity. This almost-top-down parser has then to use something like `while` loops to replace the missing recursive function calls.

To make this more concrete, if we try to directly implement the following grammar production from the original grammar:

$$E \quad \rightarrow \quad E \texttt{ + } T \mid T$$

we then get the following code:

```
match_E(){
  /* two possible productions so use lookahead token to decide */
  lookahead_token = peek_at_next_input_token();
  if(lookahead_token == '???'){
    E_node = match_E();
    p_node = match_token('+');
    T_node = match_T();
    return new_E_node(E_node, p_node, T_node);
  }else{
    T_node = match_T();
    return new_E_node(T_node);
  }
}
```

As we have seen above, depending on what `???` represents we get a piece of code that either rejects most of the strings we would like it to accept, or goes into an infinite recursive loop.

If we eliminate left recursion from the grammar production to get these new productions:

$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & \texttt{+}\ T\ E' \mid \varepsilon \end{array}$$

then the corresponding code becomes:

```
match_E(){
  /* only one production so no need for lookahead token */
  T_node = match_T();
  Ep_node = match_Ep();
  return new_E_node(T_node, Ep_node);
}

match_Ep(){
  /* two possible productions so use lookahead token to decide */
```

```
  lookahead_token = peek_at_next_input_token();
  if(lookahead_token == '+'){
    p_node = match_token('+');
    T_node = match_T();
    Ep_node = match_Ep();
    return new_Ep_node(p_node, T_node, Ep_node);
  }else{
    e_node = match_epsilon();
    return new_Ep_node(e_node);
  }
}
```

This code works fine except that the + operator is now right associative instead of being left associative (try this code on some input like T + T + T + T to see how it works). We then have only two possible solutions to solve this problem: use a bottom-up parser instead of a top-down parser, or fix the code by hand. Bottom-up parsers do not care about left recursion (or right recursion for that matter) so if we use a bottom-up parser instead of a top-down parser then we do not have to eliminate left recursion from the original grammar above, which means the associativity of + then remains the same.

If on the other hand we decide to fix the code by hand, then we need to start again from the code for the original version of the grammar:

```
match_E(){
  /* two possible productions so use lookahead token to decide */
  lookahead_token = peek_at_next_input_token();
  if(lookahead_token == '???'){
    E_node = match_E();
    p_node = match_token('+');
    T_node = match_T();
    return new_E_node(E_node, p_node, T_node);
  }else{
    T_node = match_T();
    return new_E_node(T_node);
  }
}
```

then remove the problematic recursive call to `match_E` and replace it with a `while` loop:

```
match_E(){
  T_node = match_T();
  E_node = new_E_node(T_node);
  lookahead_token = peek_at_next_input_token();
  while(lookahead_token == '+'){
    p_node = match_token('+');
    T_node = match_T();
```

```
    E_node = new_E_node(E_node, p_node, T_node);
    lookahead_token = peek_at_next_input_token();
  }
  return E_node;
}
```

In this code the recursive call to `match_E` is gone and has been replaced by a `while` loop that just reads the different T nonterminals one after the other as long as they are separated by `+` tokens (a `+` token normally always appears between two Ts). As it reads each new pair of `+` token and T nonterminal, the `while` loop builds new E parse tree nodes in the right order so that the `+` operator remains left associative in the resulting parse tree. What this code does in essence is to replace the `match_Ep` recursive function above which made the `+` operator right associative into a `while` loop that preserves the left associativity of `+`. Try to run both this code and the previous one that used `match_Ep` on some input like T `+` T `+` T `+` T and you will see that this code produces a parse tree in which `+` is left associative while the previous code that uses `match_Ep` produces a symmetric parse tree in which the `+` operator is right associative. Obviously we want the `+` operator to be left associative, since this is the normal associativity of `+` in mathematics, so we need to use in our parser the version of `match_E` that uses the `while` loop. The same technique can then be used for the `*` operator too.

(Recursive) Predictive Parsers

Now that we know how left factoring and left recursion elimination work, we can actually take an unambiguous context-free grammar and apply these two techniques to it. If the resulting grammar has such a form that a top-down recursive-descent parser can always make the right production choice at every step by only looking at most at one token of lookahead and without ever needing to backtrack, then the grammar is said to be LL(1). LL(1) means that the corresponding parser reads input from Left to right, computes a Leftmost derivation of the input sentence, and needs at most 1 token of lookahead at each step in the parsing process.

By definition, a top-down recursive-descent parser for an LL(1) grammar always selects the right grammar production at every step in the parsing process, and it does so without needing to look at more than one lookahead token at any point time and without having to backtrack. Such a parser is then called a *predictive parser*, because it can always correctly predict what it has to do next. Unless otherwise specified, a predictive parser is recursive, since every predictive parser is also a recursive-descent parser.

For every LL(1) grammar we can construct a corresponding (recursive) predictive parser: we just use the same method as above for recursive-descent parsers (since every predictive parser is just a recursive-descent parser for a restricted LL(1) grammar). We are then guaranteed that the code of the parser never needs to backtrack, never needs to use more than one token of lookahead,

and always makes the right grammar production decisions.

Note that it is not enough to do left factoring and left recursion elimination on some unambiguous context-free grammar to automatically get an LL(1) grammar. There are unambiguous CFGs which cannot be transformed into an equivalent LL(1) grammar, even after doing left factoring and left recursion elimination. For example, the unambiguous grammar for `if` statements that we presented at the end of the "Parse Trees and Ambiguity" section (the grammar that uses the M and U nonterminals) cannot be transformed into an LL(1) grammar (in fact it cannot be transformed into an LL(k) grammar for any number k of tokens of lookahead). For a grammar to be LL(1), the grammar should at a minimum be unambiguous, left factored, and without left recursion, but the shape of the grammar should also be such that the parser only ever needs at most one token of lookahead to decide at each step which production to use. Not all CFGs can be transformed to have such a shape.

Here is an example of an LL(1) grammar:

$$\begin{array}{lcl} L & \rightarrow & \texttt{id}\ T \\ T & \rightarrow & \texttt{,}\ \texttt{id}\ T \\ & | & \texttt{;} \end{array}$$

This is in fact the grammar for lists of identifiers that we used in our introduction to top-down parsing towards the beginning of these lecture notes. The reason we could parse the input string "`id , id , id ;`" using a non-backtracking parsing process in that introduction is because the grammar is LL(1) and therefore the top-down parsing process we described in that introduction was in fact a description of the behavior of the predictive parser associated with that LL(1) grammar, which only ever needed at most one lookahead token to decide which production to use at each step of the parsing.

Looking at this grammar you can see that it is left factored and does not have any left recursion. You can also see that there is no choice for the production for L, and the choice between the two productions for T can always be made by checking whether the lookahead token (the next token in the input) is either "`,`" or "`;`". The recursive-descent parser for this grammar therefore never needs more than one token of lookahead to decide which production to use at every step of the parsing process. All these properties of the grammar prove that the grammar is LL(1) and that the corresponding recursive-descent parser is a predictive parser.

Using the exact same method as in the more general case of recursive-descent parsers, we can easily create the code for this predictive parser, based directly on the shape of the grammar:

```
match_L(){
  /* only one production so no need for lookahead token */
  id_node = match_token('id');
  T_node = match_T();
  return new_L_node(id_node, T_node);
}
```

```
match_T(){
  /* two possible productions so use lookahead token to decide */
  lookahead_token = peek_at_next_input_token();
  if(lookahead_token == ','){
    c_node = match_token(',');
    id_node = match_token('id');
    T_node = match_T();
    return new_T_node(c_node, id_node, T_node);
  }else{
    s_node = match_token(';');
    return new_T_node(s_node);
  }
}
```

If you remember the grammar for arithmetic expression we talked about above in the section on left recursion elimination:

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

After eliminating left recursion we got this grammar:

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

which is in fact LL(1) too. The corresponding predictive parser can then again be directly derived from the shape of this transformed grammar:

```
match_E(){
  /* only one production so no need for lookahead token */
  T_node = match_T();
  Ep_node = match_Ep();
  return new_E_node(T_node, Ep_node);
}

match_Ep(){
  /* two possible productions so use lookahead token to decide */
  lookahead_token = peek_at_next_input_token();
  if(lookahead_token == '+'){
    p_node = match_token('+');
    T_node = match_T();
    Ep_node = match_Ep();
    return new_Ep_node(p_node, T_node, Ep_node);
```

```
  }else{
    e_node = match_epsilon();
    return new_Ep_node(e_node);
  }
}

match_T(){
  /* only one production so no need for lookahead token */
  F_node = match_F();
  Tp_node = match_Tp();
  return new_T_node(F_node, Tp_node);
}

match_Tp(){
  /* two possible productions so use lookahead token to decide */
  lookahead_token = peek_at_next_input_token();
  if(lookahead_token == '*'){
    m_node = match_token('*');
    F_node = match_F();
    Tp_node = match_Tp();
    return new_Tp_node(m_node, F_node, Tp_node);
  }else{
    e_node = match_epsilon();
    return new_Tp_node(e_node);
  }
}

match_F(){
  /* two possible productions so use lookahead token to decide */
  lookahead_token = peek_at_next_input_token();
  if(lookahead_token == '('){
    op_node = match_token('(');
    E_node = match_E();
    cp_node = match_token(')');
    return new_F_node(op_node, E_node, cp_node);
  }else{
    id_node = match_token('id');
    return new_F_node(id_node);
  }
}
```

So to write the code of a predictive parser for an LL(1) grammar is trivial: the structure of the grammar directly gives you the structure of the code.

This explains why such LL(1) grammars are often used for simple programming languages like domain-specific languages (the languages used for the configuration files of various software, for example, or used in simple network pro-

tocols like SMTP): because for such simple languages it is easy to write an LL(1) grammar, for which it is then easy to implement a simple hand-written top-down (recursive) predictive parser.

Note finally that it is not always obvious whether a given grammar is LL(1) or not. At the end of the next section we will see some mathematical conditions that we can use to decide such a question.

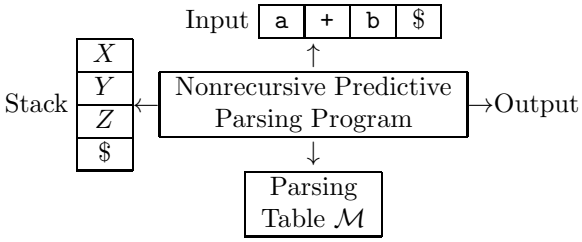
Nonrecursive Predictive Parsers

The kind of (recursive) predictive parsers we have just seen in the previous section are nice because they are very simple to implement: one just has to write recursive code that directly follows the shape of the corresponding LL(1) grammar.

The problem with this kind of (recursive) predictive parsers is that they are not very efficient. Even though such parsers are guaranteed to not do any backtracking, these parsers have to do a function call for every token and nonterminal that appear in the different grammar productions. This results in a parser that spends a lot of time calling functions and returning from functions, and which is therefore not very efficient.

Instead of using such relatively inefficient (recursive) predictive parsers, one can instead use equivalent *nonrecursive predictive parsers*. These nonrecursive predictive parsers conceptually work just like their recursive counterparts, in the sense that at each step they only need to look at most at one lookahead token to decide what to do next, and therefore never need to backtrack. So they work on exactly the same kind of LL(1) grammars as the recursive predictive parsers do. The difference is that these nonrecursive predictive parsers maintain a stack explicitly to remember where in the parsing process they are, instead of relying on the function call stack to do that implicitly like recursive predictive parsers do.

The main problem with using an explicit stack is then to determine at each step of the parsing process which production to use next. To make this decision, nonrecursive predictive parsers use a parsing table. The structure of a nonrecursive predictive parser is then as follows:



Here the input is a sequence of tokens with the special symbol \$ at the end to explicitly mark the end of the input (\$ can in fact be a special token coming from the lexer which the lexer uses to indicate that it has reached the end of the input). As we indicated above, the parser maintains an explicit stack of tokens

and nonterminals (symbols like X, Y, and Z in our drawing can each represent either a token or a nonterminal) with again the special symbol \$ to mark the bottom of the stack. The parser uses this stack to remember what it expects to see next in the input. The *parsing table* is a two-dimensional array $\mathcal{M}[A, \texttt{a}]$ where A is a nonterminal and `a` is a token or the special symbol \$.

At each step of the parsing process, the nonrecursive predictive parser looks at the next token in the input, looks at the symbol at the top of the stack, and based on that decides what to do next. In some cases the parser will then have to look at the content of the parsing table to decide which grammar production to use next. We now describe more formally this parsing algorithm.

Parsing Algorithm for Nonrecursive Predictive Parsers

To parse some input using a nonrecursive predictive parser, do the following:

- Put the special symbol \$ at the end of the input.
- Put the special symbol \$ at the bottom of the stack, and then put the start symbol of the grammar in the stack too on top of the \$.
- At each step of the parsing process, the parser considers the symbol X at the top of the stack (which can be a token, a nonterminal, or \$) and the next input symbol `a` (which can be a token or \$).
  - If both X and `a` are \$, then the parser accepts the input.
  - If both X and `a` are the same token, then remove that token from the top of the stack and consume that token in the input (i.e. move the input to the next token).
  - If X is a nonterminal, then consult $\mathcal{M}[X, \texttt{a}]$. If $\mathcal{M}[X, \texttt{a}]$ is blank, then raise a syntax error. Otherwise if $\mathcal{M}[X, \texttt{a}]$ is the grammar production $X \to UVW$, then remove X from the top of the stack, put WVU at the top of the stack (with U on top!), and output the grammar production $X \to UVW$.

The reason this algorithm puts the start symbol in the stack at the beginning is because we want to check that the input string is a valid string of the programming language (i.e. a syntactically correct, whole program) and therefore that this input string belongs to the set of strings represented by the start symbol of the grammar (which represents the set of all syntactically correct, whole programs). In general, in this algorithm, the content of the stack always indicates what the parser expects to see at various points in the remaining input. In particular the symbol at the top of the stack always indicates what the parser expects to see next in the input.

If, while the algorithm is running, both X and `a` are \$, then it means that the parser has reached the end of the input, that the bottom of the stack has been reached, and therefore that the start symbol that was just above the \$ in the stack has been removed. Since this start symbol has been removed from the

stack upon reading the whole input, it means that the input string belongs to the set of strings represented by the start symbol, i.e. the input string belongs to the set of strings described by the grammar on which the parser is based. In other words, the input string is a syntactically correct, complete program.

If, while the algorithm is running, both X and `a` are the same token, then the parser removes the token from both the top of the stack and the input. This simply means that the parser has found in the input the token that it expected to see next.

If, while the algorithm is running, X is a nonterminal and `a` is a token or \$, then the parser uses the corresponding entry in the parsing table to decide which grammar production to use to try to match such X with what comes next in the input. If the parsing table says to use a grammar production like $X \rightarrow UVW$, then the parser replaces X on the stack with WVU (U at the top!) The reason U should be at the top of the stack is because the parser reads input from left to right. This means that, using a grammar production like $X \rightarrow UVW$, what the parser expects to see next in the input is some tokens matching a U, so U should be placed at the top of the stack.

In essence, the parser is currently expecting to find in the input a sequence of tokens that matches the structure of an X. An X might have many different possible structures though (for example, a statement might be an `if` statement, or a `while` statement, or a `for` statement, etc.) By looking at the first token `a` in the input, the parser gets enough information to determine more precisely what kind of X it can expect to find in the input: an X that is of the UVW shape. Based on that the parser then replaces the relatively abstract X on the stack with the more precise expected shape UVW (for example, if the parser is currently expecting to see a statement in the input, and the next token is `while`, then the parser can remove the nonterminal for statements from the top of the stack and replace it with a more precise sequence of tokens and nonterminals that describes the expected shape of a `while` loop). Compare this with the way recursive-descent parsers use the function call stack to implicitly keep track of what they expect to see next in the input (see the section "Implementation of Recursive-Descent Parsers").

From the algorithm above, we see that the result of the parsing process is a list of grammar productions that are used by the parser when it needs to replace some nonterminal at the top of the stack. This list of grammar productions from the output of the parser in fact gives us the list of grammar productions to use in a *leftmost derivation* of the input string. This is what we expect, since we have said before that all top-down parsers (whether recursive or nonrecursive) compute a leftmost derivation of their input.

In fact we are going to see in an example below that, at every point in time in the parsing process, the list of input tokens that have already been processed, concatenated with the list of grammar symbols currently on the stack (from top to bottom), will always form together one of the sentential forms that appear in the leftmost derivation of the input.

Now that we have seen how the algorithm works, we can look at an example. Here is again the grammar for arithmetic expression, after it has been

left factored and left recursion has been eliminated (as we have said above, nonrecursive predictive parsers work on the same LL(1) grammars as recursive predictive parsers, and such LL(1) grammars should always be left factored and have no left recursion):

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

From this LL(1) grammar, we will see later how to construct a parsing table. Here I give you directly the parsing table so we can use it to parse some input:

	`id`	`+`	`*`	`(`	`)`	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow \texttt{+}\ TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow \texttt{*}\ FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \texttt{id}$			$F \rightarrow \texttt{(}\ E\ \texttt{)}$		

Using this parsing table we can then parse the following input: `id + id * id` as follows. First we put on the stack the symbol $ and the start symbol E of the grammar, which indicates the fact that the parser expects to find a complete arithmetic expression in its input. We also put a $ symbol at the end of the input.

Stack	Input	Output
$\$E$	`id + id * id` $	

The symbol at the top of the stack is a nonterminal so we know we have to use the parsing table. The nonterminal at the top of the stack is E and the next input symbol is `id`. The parsing table entry for $\mathcal{M}[E, \texttt{id}]$ is $E \rightarrow TE'$, so we remove the E from the top of the stack and replace it with $E'T$ (with T at the top of the stack, because the parser now expects to match a T with the tokens that come next in the input!) The output of the parser is the grammar production $E \rightarrow TE'$.

Stack	Input	Output
$\$E'T$	`id + id * id` $	$E \rightarrow TE'$

The nonterminal T is now at the top of the stack and `id` is still the next token in the input. The parsing table entry $\mathcal{M}[T, \texttt{id}]$ is $T \rightarrow FT'$ so we replace the T at the top of the stack with $T'F$ (F at the top of the stack!) The output of the parser is the grammar production $T \rightarrow FT'$.

Stack	Input	Output
$\$E'T'F$	`id + id * id` $	$T \rightarrow FT'$

The nonterminal F is now at the top of the stack and `id` is still the next token in the input. The parsing table entry $\mathcal{M}[F, \texttt{id}]$ is $F \to \texttt{id}$, so we replace the F at the top of the stack with `id`. The output of the parser is the grammar production $F \to \texttt{id}$.

Stack	Input	Output
$\$E'T'\texttt{id}$	`id + id * id $`	$F \to \texttt{id}$

The symbol at the top of the stack is now a token. This token at the top of the stack indicates which token the parser expects to find next in the input. If the token at the top of the stack matches the next token in the input then we remove them both (i.e. the parser consumes the input token). If the token at the top of the stack does not match the next token in the input then we have a syntax error. In the present case both tokens at the top of the stack and in the input are `id`, so we remove them:

Stack	Input	Output
$\$E'T'$	`+ id * id $`	

The nonterminal T' is now at the top of the stack and `+` is the next token in the input. The parsing table entry $\mathcal{M}[T', \texttt{+}]$ is $T' \to \varepsilon$, so we replace the T' at the top of the stack with nothing (the empty string). The output of the parser is the grammar production $T' \to \varepsilon$.

Stack	Input	Output
$\$E'$	`+ id * id $`	$T' \to \varepsilon$

The nonterminal E' is now at the top of the stack and `+` is still the next token in the input. The parsing table entry $\mathcal{M}[E', \texttt{+}]$ is $E' \to \texttt{+}\ TE'$, so we replace the E' at the top of the stack with $E'T\texttt{+}$ (`+` at the top of the stack!). The output of the parser is the grammar production $E' \to \texttt{+}\ TE'$.

Stack	Input	Output
$\$E'T\texttt{+}$	`+ id * id $`	$E' \to \texttt{+}\ TE'$

This whole process repeats itself until we reach this point:

Stack	Input	Output
$\$E'T'\texttt{id}$	`id $`	$F \to \texttt{id}$

Here the token `id` is both at the top of the stack and next in the input so we remove both:

Stack	Input	Output
$\$E'T'$	$	

The nonterminal T' is now at the top of the stack and the special symbol \$ is next in the input. The parsing table entry $\mathcal{M}[T', \$]$ is $T' \to \varepsilon$, so we replace the T' at the top of the stack with nothing. The output of the parser is the grammar production $T' \to \varepsilon$.

Stack	Input	Output
$\$E'$	\$	$T' \to \varepsilon$

The nonterminal E' is now at the top of the stack and the special symbol \$ is still next in the input. The parsing table entry $\mathcal{M}[E', \$]$ is $E' \to \varepsilon$, so we replace the E' at the top of the stack with nothing. The output of the parser is the grammar production $E' \to \varepsilon$.

Stack	Input	Output
\$	\$	$E' \to \varepsilon$

At this point the special symbol \$ appears both at the top of the stack and next in the input. So the parsing algorithm stops and the parser accepts the input string. In other words, the input string `id + id * id` is a correct sentence in the programming language described by the grammar:

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

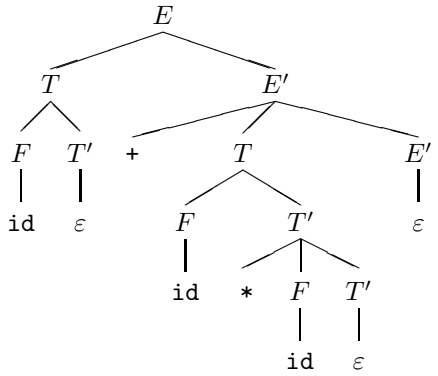
Here the complete result of the parsing algorithm:

Stack	Input	Output
$\$E$	`id + id * id` \$	
$\$E'T$	`id + id * id` \$	$E \to TE'$
$\$E'T'F$	`id + id * id` \$	$T \to FT'$
$\$E'T'$`id`	`id + id * id` \$	$F \to$ `id`
$\$E'T'$	`+ id * id` \$	
$\$E'$	`+ id * id` \$	$T' \to \varepsilon$
$\$E'T$`+`	`+ id * id` \$	$E' \to$ `+` TE'
$\$E'T$	`id * id` \$	
$\$E'T'F$	`id * id` \$	$T \to FT'$
$\$E'T'$`id`	`id * id` \$	$F \to$ `id`
$\$E'T'$	`* id` \$	
$\$E'T'F$`*`	`* id` \$	$T' \to$ `*` FT'
$\$E'T'F$	`id` \$	
$\$E'T'$`id`	`id` \$	$F \to$ `id`
$\$E'T'$	\$	
$\$E'$	\$	$T' \to \varepsilon$
\$	\$	$E' \to \varepsilon$

From the list of grammar productions in the output of the algorithm, we can easily construct the leftmost derivation of the input sentence:

	E	start symbol
$\Rightarrow$	TE'	$E \to TE'$
$\Rightarrow$	$FT'E'$	$T \to FT'$
$\Rightarrow$	`id` $T'E'$	$F \to$ `id`
$\Rightarrow$	`id` E'	$T' \to \varepsilon$
$\Rightarrow$	`id +` TE'	$E' \to$ `+` TE'
$\Rightarrow$	`id +` $FT'E'$	$T \to FT'$
$\Rightarrow$	`id + id` $T'E'$	$F \to$ `id`
$\Rightarrow$	`id + id *` $FT'E'$	$T' \to$ `*` FT'
$\Rightarrow$	`id + id * id` $T'E'$	$F \to$ `id`
$\Rightarrow$	`id + id * id` E'	$T' \to \varepsilon$
$\Rightarrow$	`id + id * id`	$E' \to \varepsilon$

and from this leftmost derivation we can easily construct the parse tree:



As we said above, at every point in time in the parsing process, the list of input tokens that have already been processed, concatenated with the list of grammar symbols currently on the stack (read in the present case from right to left) will always form together one of the sentential forms that appear in the leftmost derivation of the input. So, for example, at the following point in the parsing process:

Stack	Input	Output
$\$E'T'$`id`	`id * id` $	$F \to$ `id`

the parser has already processed the tokens `id +` (i.e. only the tokens `id * id` remain in the input). If you concatenate these two tokens `id +` with the content of the stack at that point in time (read from right to left): `id` $T'E'$ then you get the sequence of grammar symbols `id + id` $T'E'$, which is exactly the sentential form at the corresponding step in the leftmost derivation above. You can do the same for all steps in the parsing process above, which clearly shows that the nonrecursive predictive parser computes a leftmost derivation.

Implementing Nonrecursive Predictive Parsers

Given the parsing algorithm above, it is easy to implement the code of a nonrecursive predictive parser: use a one-dimensional array to implement the stack, use a two-dimensional array to implement the parsing table (in which each grammar production can simply be represented by its corresponding number in the grammar), then write C code that directly follows the algorithm by looking at what is 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 directly build the corresponding parse tree by creating new parse tree nodes based on the grammar productions used, instead of outputting the grammar productions. This is not very different from the way we create parse tree nodes in the C code for recursive-descent parsers and (recursive) predictive parsers, so again this is left as an exercise to the reader.

The hard part in implementing a nonrecursive predictive parser is therefore not writing the actual code, but rather first computing which grammar productions should appear where in 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 Nonrecursive Predictive Parsers

Remember that a predictive parser is always able to decide which production to use by looking at most at one token of lookahead.

Suppose that the grammar contains a production of the form $A \to \alpha$, where α is some sequence of tokens and nonterminals. Suppose also that some of the strings in the set of strings represented by α start with the token `a`. Then if the nonterminal A is at the top of the stack and the token `a` is next in the input (i.e. `a` is the lookahead token), we want the parser to use the grammar production $A \to \alpha$ to replace A at the top of the stack with α (read backwards, so that the beginning of α appears at the top of the stack!) So the idea to construct the parsing table is then simply to put the grammar production $A \to \alpha$ in the table entry $\mathcal{M}[A, \texttt{a}]$, and to do this for all tokens `a` that can start strings in the set of strings represented by α.

From this it appears that the only thing we really have to do is to compute the list of tokens `a` that can start strings in the set of strings represented by α, and then put the grammar production $A \to \alpha$ at the right places in the parsing table. This is in fact part of what we are going to have to do: for each sequence of grammar symbols α that appears in the grammar, we are going to compute the set FIRST(α) that tells us which tokens can appear at the start of any string in α.

There is one complication though (of course!) Suppose we have a grammar production like $A \to BC$. Then is seems logical to assume that any token that can appear at the start of B can also appear at the start of A. But if in addition

there is a production of the form $B \to \varepsilon$, then we also have to take into account the fact that any token that can appear at the start of C can also appear at the start of A (since B can essentially be made to disappear by deriving the empty string). This means that what can *follow* the strings in B can also start the strings in A. To take this problem into account, we are also going to have to compute sets FOLLOW(A) for each nonterminal A that appears in the grammar for which we want to construct a parsing table...

Of course all this should be computed using an LL(1) grammar which has been left factored and from which left recursion has been eliminated, since predictive parsers (whether recursive or nonrecursive) only work with such grammars.

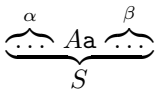
Here are the more formal definitions for FIRST and FOLLOW:

- If α is a string of grammar symbols (i.e. any sequence of tokens and nonterminals), then FIRST(α) is the set of tokens that begin all the possible strings derived from α: FIRST(α) = $\{\mathtt{a} \mid \alpha \stackrel{*}{\Rightarrow} \mathtt{a}\beta\}$. If $\alpha \stackrel{*}{\Rightarrow} \varepsilon$, then ε is also in FIRST(α).
- If A is a nonterminal, then FOLLOW(A) is the set of tokens $\mathtt{a}$ that can appear immediately to the right of A (i.e. immediately follow A) in some sentential form: FOLLOW(A) = $\{\mathtt{a} \mid S \stackrel{*}{\Rightarrow} \alpha A \mathtt{a} \beta\}$. If A can be the rightmost symbol in some sentential form, then \$ is in FOLLOW(A).

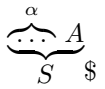
In other words, if you think about some string of grammar symbols α as representing a set of input strings, then FIRST(α) is the set of all the input tokens that start some input strings in α. So if $\alpha \stackrel{*}{\Rightarrow} \mathtt{a}\ \beta$, for some sequence of grammar symbols β, then $\mathtt{a}$ is in FIRST(α).

If $\alpha \stackrel{*}{\Rightarrow} \varepsilon$, then α can be nothing, so it can start with nothing, so ε is in FIRST(α).

If there is some derivation $S \stackrel{*}{\Rightarrow} \alpha\ A\ \mathtt{a}\ \beta$, for some strings of grammar symbols α and β (and where S is the start symbol of the grammar), then $\mathtt{a}$ is in FOLLOW(A).



If there is some derivation $S \stackrel{*}{\Rightarrow} \alpha\ A$, for some string of grammar symbols α (and where S is the start symbol of the grammar), then \$ is in FOLLOW(A).



To put it in yet another (simpler) way, FIRST(X) tells the parser which tokens it can expect next in the input when starting to match an X, and FOLLOW(X) tells the parser which tokens it can expect next in the input after having matched a complete X.

For example, given the LL(1) grammar for arithmetic expressions that we have used before, and the input sentence "`( id ) + ...`", we can match the first part "`( id )`" of the sentence as an F. Then "`(`" is in FIRST(F), since the F starts with "`(`", and "`+`" is in FOLLOW(F), since "`+`" appears in the input sentence right after the part of the sentence that is matched by the F.

These definitions are really nice, but they do not tell us how to actually compute FIRST and FOLLOW, so here are algorithms.

- If X is a single grammar symbol, repeat the following rules until no set changes anymore:
  - If X is a token, then FIRST(X) is $\{X\}$.
  - If X is a nonterminal and $X \to \varepsilon$, then add ε to FIRST(X).
  - If X is a nonterminal and $X \to Y_1\ Y_2\ \ldots\ Y_k$, then FIRST($Y_1$) - ε is in FIRST(X). If in addition ε is in FIRST(Y_1) (in other words, $Y_1 \stackrel{*}{\Rightarrow} \varepsilon$) then FIRST($Y_2$) - ε is in FIRST(X). If in addition ε is in FIRST(Y_2) (in other words, $Y_1 Y_2 \stackrel{*}{\Rightarrow} \varepsilon$) then FIRST($Y_3$) - ε is in FIRST(X). Etc. In general, if ε is in all the sets FIRST(Y_1)...FIRST(Y_{i-1}) (in other words, $Y_1\ \ldots\ Y_{i-1} \stackrel{*}{\Rightarrow} \varepsilon$) then FIRST($Y_i$) - ε is in FIRST(X). Finally, if $X \stackrel{*}{\Rightarrow} \varepsilon$, then ε is in FIRST(X).
- If X is a sequence of grammar symbols $X_1\ X_2\ \ldots\ X_n$, then FIRST(X_1) - ε is in FIRST(X). If in addition ε is in FIRST(X_1) (in other words, $X_1 \stackrel{*}{\Rightarrow} \varepsilon$) then FIRST($X_2$) - ε is in FIRST(X). If in addition ε is in FIRST(X_2) (in other words, $X_1 X_2 \stackrel{*}{\Rightarrow} \varepsilon$) then FIRST($X_3$) - ε is in FIRST(X). Etc. In general, if ε is in all the sets FIRST(X_1)...FIRST(X_{i-1}) (in other words, $X_1\ \ldots\ X_{i-1} \stackrel{*}{\Rightarrow} \varepsilon$) then FIRST($X_i$) - ε is in FIRST(X). Finally, if $X \stackrel{*}{\Rightarrow} \varepsilon$, then ε is in FIRST(X).

The first rule means that a token can only start with itself. The second rule means that, if a nonterminal X can be the empty string, then an input string matching X can start with nothing (i.e. be the empty string). The third and fourth rules are very similar: whatever can start the first symbol in a sequence of symbols, can start the whole sequence; if the first symbol can be the empty string then whatever can start the second symbol in the sequence can also start the whole sequence; if the second symbol can also be the empty string (in addition to the first one also possibly being the empty string) then whatever can start the third symbol in the sequence can also start the whole sequence; etc.

Note that the fourth rule is vacuously true when the sequence is empty (i.e. $n = 0$). In other words, FIRST(ε) is $\{\varepsilon\}$.

To compute FOLLOW(A) for all nonterminals A, apply the following rules until no set changes anymore:

- If S is the start symbol of the grammar, then put \$ in FOLLOW(S).
- If $A \to \alpha\ B\ \beta$, then FIRST(β) - ε is in FOLLOW(B).
- If $A \to \alpha\ B$, or $A \to \alpha\ B\ \beta$ where $\beta \stackrel{*}{\Rightarrow} \varepsilon$ (i.e. ε is in FIRST(β)), then FOLLOW(A) is in FOLLOW(B).

The first rule means that, since the start symbol of a grammar represents complete programs, then what we expect to see after a complete program is the end of the input. The second rule means that, if some β can follow B, then whatever token can start β can follow B. The third rule means that, if an A can end with a B (either directly from a grammar production, or because some β that follows B can become the empty string) then whatever can follow A can also follow B.

These rules seem fairly complex, so lets look at an example of how they work. Again we are going to use the left factored, non-left recursive grammar for arithmetic expressions:

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

We first compute the FIRST sets for the tokens. For each token in the grammar the corresponding FIRST set is simply the set containing only that token:

X	`id`	`+`	`*`	`(`	`)`
FIRST(X)	{`id`}	{`+`}	{`*`}	{`(`}	{`)`}

We now have to compute the FIRST sets for the nonterminals. One rule for computing the FIRST sets for nonterminals says that, if $X \to \varepsilon$ is a grammar production, then ε should be in FIRST(X). Therefore ε is in both FIRST(E') and FIRST(T'):

X	E	E'	T	T'	F
FIRST(X)		$\{\varepsilon\}$		$\{\varepsilon\}$	

Another rule for computing the FIRST sets for nonterminals says that, if $X \to Y_1\ Y_2 \ldots Y_k$ is a grammar production, and if ε is in all the sets FIRST(Y_1), ..., FIRST(Y_{i-1}) (in other words, if $Y_1 \ldots Y_{i-1} \stackrel{*}{\Rightarrow} \varepsilon$) then FIRST($Y_i$) - ε is in FIRST(X).

In the present case, only E' and T' can derive ε, and E' and T' never appear as the leftmost nonterminal of the RHS of a grammar production (in fact, both E' and T' only ever appear as the rightmost nonterminal in the grammar productions where they appear) so the third rule just above can be simplified into: if $X \to Y_1\ Y_2\ \ldots Y_k$ is a grammar production then FIRST(Y_1) - ε is in FIRST(X) (i.e., whatever can appear at the beginning of Y_1 can also appear at the beginning of X).

If we apply this rule to each of the grammar productions for the nonterminal F, then it means that `(` and `id` are both in FIRST(F). This is obvious, since any sentence that can be derived from the nonterminal F has to start with one of those two tokens. So we get the following table:

X	E	E'	T	T'	F
FIRST(X)		$\{\varepsilon\}$		$\{\varepsilon\}$	{`(`, `id`}

If we apply the same rule to the production $T' \to$ `*` FT', then whatever is in FIRST(`*`) - ε should also be in FIRST(T'). Since `*` is the sole element in FIRST(`*`), we add `*` to FIRST(T'). Similarly for FIRST(E') with `+`:

X	E	E'	T	T'	F
FIRST(X)		$\{\varepsilon$, `+`$\}$		$\{\varepsilon$, `*`$\}$	{`(`, `id`}

The last two grammar productions that we have not considered yet are $T \to FT'$ and $E \to TE'$. Again we apply the same rule, which, from the grammar production $T \to FT'$, tells us that whatever is in FIRST(F) - ε is also in FIRST(T), and which, from the grammar production $E \to TE'$, tells us that whatever is in FIRST(T) - ε is also in FIRST(E). We therefore obtain the final sets for the nonterminals:

X	E	E'	T	T'	F
FIRST(X)	{`(`, `id`}	$\{\varepsilon$, `+`$\}$	{`(`, `id`}	$\{\varepsilon$, `*`$\}$	{`(`, `id`}

We now have to compute the FOLLOW sets. The first rule to compute the FOLLOW sets tells us to put \$ in the set FOLLOW(E) since E is the start symbol of the grammar:

X	E	E'	T	T'	F
FOLLOW(X)	{\$}				

The second rule to compute the FOLLOW sets tells us that, for each grammar production of the form $A \to \alpha\ B\ \beta$, everything in FIRST(β) - ε is also in FOLLOW(B) (i.e., whatever can start β can follow B, since β appears in the grammar production right after B).

If we apply this rule to the $F \to$ `(` E `)` grammar production, then we conclude that FIRST(`)`) - ε is in FOLLOW(E):

X	E	E'	T	T'	F
FOLLOW(X)	{\$, `)`}				

By applying the same rule to the $T \rightarrow FT'$ grammar production (or, equally, the $T' \rightarrow$ * FT' grammar production), we conclude that FIRST(T') - ε is in FOLLOW(F). Similarly, from the $E \rightarrow TE'$ grammar production (or, equally, the $E' \rightarrow$ + TE' grammar production), we conclude that FIRST(E') - ε is in FOLLOW(T). So we get:

X	E	E'	T	T'	F
FOLLOW(X)	{$,)}		{+}		{*}

The third rule for computing the FOLLOW sets tells us first that, for each grammar production of the form $A \rightarrow \alpha\ B$, everything in FOLLOW(A) is also in FOLLOW(B). If we apply this rule to the production $E \rightarrow TE'$, then we conclude that FOLLOW(E) is in FOLLOW(E'). Similarly, the grammar production $T \rightarrow FT'$ tells us that everything in FOLLOW(T) is also in FOLLOW(T').

Finally, the second part of the third rule for computing the FOLLOW sets tells us that, for each grammar production of the form $A \rightarrow \alpha\ B\ \beta$, if ε is in FIRST(β) (i.e., $\beta \stackrel{*}{\Rightarrow} \varepsilon$), then everything in FOLLOW(A) is again also in FOLLOW(B).

The only nonterminals with ε in their FIRST set are E' and T', so to apply this second part of the third rule to compute the FOLLOW sets we only have to look at the grammar productions that end with E' or T'. From grammar production $E \rightarrow TE'$ we therefore conclude that FOLLOW(E) is in FOLLOW(T). From grammar production $E' \rightarrow$ + TE' we conclude that FOLLOW(E') is in FOLLOW(T). From grammar production $T \rightarrow FT'$ we conclude that FOLLOW(T) is in FOLLOW(F). And from grammar production $T' \rightarrow$ * FT' we conclude that FOLLOW(T') is in FOLLOW(F).

To summarize, the third rule for computing the FOLLOW sets tells us that:

$$\begin{array}{rcl} \text{FOLLOW}(E) & \subseteq & \text{FOLLOW}(E') \\ \text{FOLLOW}(T) & \subseteq & \text{FOLLOW}(T') \\ \text{FOLLOW}(E) & \subseteq & \text{FOLLOW}(T) \\ \text{FOLLOW}(E') & \subseteq & \text{FOLLOW}(T) \\ \text{FOLLOW}(T) & \subseteq & \text{FOLLOW}(F) \\ \text{FOLLOW}(T') & \subseteq & \text{FOLLOW}(F) \end{array}$$

or in other words:

$$\text{FOLLOW}(E) \subseteq \text{FOLLOW}(E') \subseteq \text{FOLLOW}(T) \subseteq \text{FOLLOW}(T') \subseteq \text{FOLLOW}(F)$$

We therefore have now just to copy all the sets from left to right in the table above to get the final table:

X	E	E'	T	T'	F
FOLLOW(X)	{$,)}	{$,)}	{+, $,)}	{+, $,)}	{*, +, $,)}

Once we have computed the FIRST and FOLLOW sets, we can now use these sets to create a parsing table. Remember from above that the basic idea is to put in the parsing table the production $A \rightarrow \alpha$ in all entries $\mathcal{M}[A, \mathtt{a}]$ such that a is in FIRST(α). There are also going to be a few extra rules to deal with

ε. Here is the precise algorithm to build the parsing table, based on the sets FIRST and FOLLOW we just computed.

For each production $A \to \alpha$ in the (left factored, non-left recursive) grammar, do the following:

- For each token `a` in FIRST(α), add the grammar production $A \to \alpha$ to $\mathcal{M}[A, \texttt{a}]$.
- If ε is in FIRST(α) then, for each token `b` in FOLLOW(A), add $A \to \alpha$ to $\mathcal{M}[A, \texttt{b}]$.
- If ε is in FIRST(α) and \$ is in FOLLOW(A) then add $A \to \alpha$ to $\mathcal{M}[A, \$]$.
- All other entries in the table are left blank and correspond to a syntax error.

The first rule says that, if A is at the top of the stack in the parser, and `a` is the next input token, then the parser uses the production $A \to \alpha$ (since α can start with `a`). In essence the parser looks at the lookahead token `a` and based on that decides which production to use next in the parsing process.

The second rule says that, if A is at the top of the stack in the parser, and the next input token is a `b`, which is the kind of tokens that the parser only expects to see after having matched an A in the input, then the parser uses the production $A \to \alpha$. Since $\alpha \stackrel{*}{\Rightarrow} \varepsilon$, we then expect the parser to transform the α into ε (without consuming any input) to match A with the empty string, at which point the parser will then start processing again the next tokens in the input starting with `b`.

The third rule says that, if A is at the top of the stack in the parser, and the next input symbol is \$, which is the kind of symbol that the parser only expects to see after having matched an A in the input, then the parser uses the production $A \to \alpha$. Since $\alpha \stackrel{*}{\Rightarrow} \varepsilon$, we then expect the parser to transform the α into ε (without consuming any input) to match A with the empty string, at which point the parser will then reach the end of the input and hopefully accept the whole string. This rule is in effect the exact same rule as the previous one, except specialized for the case of the special symbol \$.

So if we go back to our example, we have computed above the FIRST and FOLLOW sets for it, so we can now use these sets and the three rules above to finally construct the parsing table. To do that, we simply systematically apply the three rules to each production in the grammar one by one to get the table.

The first rule says that, for each production $A \to \alpha$ and for each token `a` in FIRST(α) (note that ε is not a token), we add the production $A \to \alpha$ to $\mathcal{M}[A, \texttt{a}]$, where M is the parsing table we are constructing.

First, let's consider only the grammar productions that do not involve ε. Then, for example, for the first grammar production $E \to TE'$, we have FIRST(T) is {`(`, `id`}, so, according to the first rule above, we put that grammar production into the two table cells that have E as nonterminal and `(` or `id` as token:

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'						
T						
T'						
F						

Similarly for the productions $E' \to$ `+` TE', $T \to FT'$, $T' \to$ `*` FT', $F \to$ `(` E `)`, and $F \to$ `id`:

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to$ `+` TE'				
T	$T \to FT'$			$T \to FT'$		
T'			$T' \to$ `*` FT'			
F	$F \to$ `id`			$F \to$ `(` E `)`		

The second rule for parse table construction says that, for each production $A \to \alpha$, if $\varepsilon \in$ FIRST(α), then for each token `b` in FOLLOW(A) (note that \$ is not a token in this regard), we add the production $A \to \alpha$ to $\mathcal{M}[A,$ `b`$]$.

None of the grammar productions $E \to TE'$, $E' \to$ `+` TE', $T \to FT'$, $T' \to$ `*` FT', $F \to$ `(` E `)`, and $F \to$ `id` have ε in FIRST(α), so this second rule does not apply to these grammar productions.

The third rule for parse table construction says that, for each production $A \to \alpha$, if $\varepsilon \in$ FIRST(α) and if \$ $\in$ FOLLOW(A), we add the production $A \to \alpha$ to $\mathcal{M}[A,$ \$$]$. This rule is very similar to the second rule above, except with \$ in place of the token `b`.

Again, since none of the grammar productions $E \to TE'$, $E' \to$ `+` TE', $T \to FT'$, $T' \to$ `*` FT', $F \to$ `(` E `)`, and $F \to$ `id` have ε in FIRST(α), this third rule does not apply to these grammar productions.

So we are done with processing all the following grammar productions, which do not involve ε: $E \to TE'$, $E' \to$ `+` TE', $T \to FT'$, $T' \to$ `*` FT', $F \to$ `(` E `)`, and $F \to$ `id`.

Now we are left with processing the two grammar productions $E' \to \varepsilon$ and $T' \to \varepsilon$. Let's consider $E' \to \varepsilon$ first. According to the rules to compute FIRST(α) when α is a sequence of symbols, we have FIRST(ε) = $\{\varepsilon\}$. This set therefore does not contain any token and the first rule we have described at the beginning of this question does not apply. Since $\varepsilon \in$ FIRST(α), the second rule does apply though. The set FOLLOW(E') is {\$, `)`}, so, based on the second rule above, we add $E' \to \varepsilon$ to $\mathcal{M}[E',$ `)`$]$. Based on the third rule above we also add $E' \to \varepsilon$ to $\mathcal{M}[E',$ \$$]$.

Similarly, for the $T' \to \varepsilon$ grammar production: since FOLLOW(T') is {`+`, \$, `)`}, we add the the grammar production $T' \to \varepsilon$ to $\mathcal{M}[T',$ `+`$]$ and $\mathcal{M}[T',$ `)`$]$, based on the second rule above, and to $\mathcal{M}[T',$ \$$]$, based on the third rule above.

The final parse table is therefore as follows:

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to \texttt{+}\ TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to \texttt{*}\ FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to \texttt{id}$			$F \to \texttt{(}\ E\ \texttt{)}$		

This parsing table is, unsurprisingly, the same table that we used when we looked at how to use the parsing algorithm to parse some input.

Creating the Parsing Table for Nonrecursive Predictive Parsers: the Summary

To create the parsing table for a nonrecursive predictive parser, do the following:

- Eliminate any ambiguity from the grammar.
- Eliminate left recursion from the grammar.
- Left factor the grammar.
- Compute the FIRST sets for all tokens and nonterminals in the grammar.
- Compute the FOLLOW sets for all nonterminals in the grammar.
- Use those FIRST and FOLLOW sets to construct the parsing table.

LL(1) Grammars

So far we have defined the concept of LL(1) grammars only in vague terms, like "LL(1) means that the corresponding parser reads input from Left to right, computes a Leftmost derivation of the input sentence, and needs at most 1 token of lookahead at each step in the parsing process" or like "the grammar should at a minimum be unambiguous, left factored, and without left recursion, but the shape of the grammar should also be such that the parser only ever needs at most one token of lookahead to decide at each step which production to use".

Now that we know how to construct a parsing table for a nonrecursive predictive parser, we are ready to give a more formal definition of LL(1) grammars.

First, note that the method we have used above to construct the parsing table can in fact be applied to *any* context-free grammar. For some grammars however, the parsing table will end up with some entries containing more than one grammar production. For example, if the grammar is ambiguous, or if the grammar is unambiguous but is not left-factored, then the parsing table will be in fact guaranteed to have at least one entry that contains multiple productions (try it on an example!)

Obviously if some entries in the parsing table contain more than one production, then we will not be able to use this parsing table in a predictive parser, since, when reaching such an entry, the predictive parser would no longer be

able to decide what to do next to parse the input (i.e. the predictive parser would stop being predictive...)

This then gives us the formal definition of LL(1) grammars: if we construct the parsing table for a grammar using the method above, and if the resulting parsing table has no entry that contains more than one grammar production, then by definition the grammar is LL(1).

In other words, an *LL(1) grammar* is by definition a grammar for which the parsing table construction method above actually works! Note that, as we have indicated before, there are some unambiguous, left-factored, non-left-recursive context-free grammars which are not LL(1) and for which, therefore, the method above will give an ambiguous parsing table. This just means that the method above is not powerful enough to handle all unambiguous grammars (like, for example, the unambiguous grammar for `if` statements that appears at the end of the "Parse Trees and Ambiguity" section above). At least we have a guarantee that, if a grammar is ambiguous, then the parsing table construction method above will always detect it.

This is a nice definition but it is not very useful to decide in advance which grammar is LL(1) and which grammar is not. We do not want to have to go through the whole parsing table construction method just to discover at the end that the grammar we started with is not LL(1)!

For simple cases it is easy to decide whether a grammar is LL(1) or not. For example, every ambiguous grammar is by definition not LL(1). Similarly, every left recursive grammar is by definition not LL(1).

There are in fact also a few conditions that we can check to prove that a given grammar *is* LL(1), without having to construct the parsing table: a grammar G is LL(1) if and only if, whenever we have two productions in G of the form $A \to \alpha \mid \beta$, the following conditions are all true:

- Strings in α and strings in β never start with the same token (or, in other words, the intersection of FIRST(α) and FIRST(β) is empty).
- At most one of α and β can derive ε (or, in other words, ε is in at most one of FIRST(α) or FIRST(β), but never in both).
- If $\beta \stackrel{*}{\Rightarrow} \varepsilon$, then no string in α can start with the same token as a string that can appear after A (in other words, the intersection of FIRST(α) and FOLLOW(A) is empty).

These conditions in essence mean that, by looking at a single token of lookahead, the predictive parser can always decide whether to use the production $A \to \alpha$ or whether to use the production $A \to \beta$, which is in fact how we informally defined LL(1) grammars at the beginning on the section on top-down parsers. So these conditions simply formally define what it means for a predictive parser to always know what to do!

Error Recovery For Nonrecursive Predictive Parsers

In this section we will limit our discussion of error recovery to nonrecursive predictive parsers, since then error recovery can be done by manipulating the parsing table for the parser. Error recovery can obviously also be done for recursive-descent parser and (recursive) predictive parsers, but we will not look at those parsers in this section.

Just like for lexical analysis, we want syntax analysis to be able to recover from (syntax) error so that the compiler can continue processing the source program and maybe discover more errors in it.

A nonrecursive predictive parser detects a syntax error when either the token at the top of the stack does not match the next token in the input, or when the parser tries to use en entry in the parsing table which is empty (i.e. no grammar production can match the input).

Once a syntax error has been discovered, the simplest way for the parser to recover from it is to start ignoring either part of the stack or part of the input (or even parts of both!) There is no mathematical rule that can tell the parser whether it should ignore things in the stack or rather ignore things in the input, and how much. Just like in the lexer case, the parser cannot be certain of what the user really meant to express in his wrong source program, so at best the parser can only try to guess what the most likely cause of the error was (a missing token? one token too many? tokens in the wrong order?) and try to guess what the best way to solve such error might be.

Here are a few things that a predictive parser can do when it detects a syntax error:

- If the entry $\mathcal{M}[A, \texttt{a}]$ is blank, that it means the parser is expecting to match an A with some input but has instead found an input token `a` that cannot begin such an A. One simple thing to do then is to simply ignore the input token `a`. Input tokens like this are ignored one after the other until either the parser comes across a parsing table entry which contains a grammar production, in which case the normal parsing process restarts from there, or until the parser sees in the input a token that can follow an A (i.e. a token in FOLLOW(A)). In that latter case the parser can then just pretend that the A at the top of the stack has actually been matched with all the input tokens that were ignored, pop A from the stack, and then go back to the regular parsing process with the next input token (which can correctly follow the A that the parser just pretended to match and popped). To implement this second case, we just have to put a "pop nonterminal" command in each entry in the table of the form $\mathcal{M}[A, \texttt{b}]$ where `b` is a token (or \$) in FOLLOW(A). All the other remaining entries then correspond to ignoring ("skipping") the next input token.

- If the token at the top of the stack does not match the next token in the input, then the nonrecursive predictive parser can simply remove the token from the top of the stack (i.e pretend that that token was actually matched by a nonexistent token in the input) and try again. This usually works

well when the user has simply forgotten to write some piece of syntax in his source program (like missing a semicolon, for example).

Based on such an error recovery strategy, the parsing table then becomes:

	`id`	`+`	`*`	`(`	`)`	$
E	$E \to TE'$	skip	skip	$E \to TE'$	pop	pop
E'	skip	$E' \to$ `+` TE'	skip	skip	$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$	pop	skip	$T \to FT'$	pop	pop
T'	skip	$T' \to \varepsilon$	$T' \to$ `*` FT'	skip	$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to$ `id`	pop	pop	$F \to$ `(` E `)`	pop	pop

with the extra rule that unmatched tokens at the top of the stack are popped too.

We can try to use this parsing table on some erroneous input like `+ id * + id`:

Stack	Input	Output
$\$E$	`+ id * + id` $	
$\$E$	`id * + id` $	skip `+`
$\$E'T$	`id * + id` $	$E \to TE'$
$\$E'T'F$	`id * + id` $	$T \to FT'$
$\$E'T'$`id`	`id * + id` $	$F \to$ `id`
$\$E'T'$	`* + id` $	
$\$E'T'F$`*`	`* + id` $	$T' \to$ `*` FT'
$\$E'T'F$	`+ id` $	
$\$E'T'$	`+ id` $	pop F
$\$E'$	`+ id` $	$T' \to \varepsilon$
$\$E'T$`+`	`+ id` $	$E' \to$ `+` TE'
$\$E'T$	`id` $	
$\$E'T'F$	`id` $	$T \to FT'$
$\$E'T'$`id`	`id` $	$F \to$ `id`
$\$E'T'$	$	
$\$E'$	$	$T' \to \varepsilon$
$	$	$E' \to \varepsilon$

Obviously trying to create a leftmost derivation and a parse tree based on the list of grammar productions computed by the predictive parser would result in a strange parse tree, that would have the first token `+` not connected to the root of the parse tree, and the nonterminal F appearing as one of the leaves of the parse tree. Both things would make such a parse tree wrong. Remember that here the goal for the syntax analysis is not to try to fix the user's source program to transform it into a correct source program with a correct parse tree. The goal is simply to be able to recover from one error to continue parsing more input. We can see that in the example just above error recovery actually works nicely, since after discovering the first error in the input (corresponding to the "skip"

and the extraneous + token at the start of the input) the parser can recover and continue parsing more input to discover the second error (corresponding to the "pop" and the extraneous ∗ token in the input).