Compiled on 08/05/2023 at 11:16am

Top-Down Parsing: The LL(1) Algorithm

We ended the previous module with the declaration that we are interested in obtaining derivations, since derivations uniquely specify the parse tree that represents the program's structure. In this module, we will cover the LL(1) top-down parsing algorithm for obtaining a derivation for an input string given a CFG. (The meaning of the name LL(1) will be explained later, once we have introduced the basic ideas of the algorithm.)

In a top-down parsing algorithm we begin at the start symbol of the grammar and apply production rules until we have obtained the input string. The name "top-down parsing" refers to the fact that we are beginning at the top of the parse tree and working our way down, towards the leaves of the tree which represent the terminals in the input string.

Given a CFG $G = (N, T, P, S)$ and a terminal string $s \in T^*$, we will begin applying production rules to S, the start symbol of the grammar, and choose rules to show that that $S \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_{n-1} \Rightarrow s$ or show that $s \notin L(G)$. Recall also from our discussion of derivation styles from the previous module; to ensure uniqueness of derivations we must fix our derivation style. We will choose leftmost derivations, i.e., whenever we are at some α_i which contains multiple non-terminals, we will choose a rule for the leftmost non-terminal.

1 Augmented Grammars

To simplify the specification of parsing algorithms, it helps if the start symbol has only one production rule. If a grammar does not satisfy this condition, we can *augment* the grammar by creating $G' = (N', T', P', S')$ from our original CFG $G = (N, T, P, S)$ where

$$\begin{aligned} N' &= N \cup \{S'\} \\ T' &= T \cup \{\vdash, \dashv\} \\ P' &= P \cup \{S' \rightarrow \vdash S \dashv\} \end{aligned}$$

The symbols, $\vdash$ and $\dashv$ symbolize the beginning and end of the file respectively. In general, the end of the file marker is especially useful since it is a symbol that is not considered part of the input. When algorithms read this symbol, they know that the end of the input has been reached.

2 An Informal Top-Down Parsing Algorithm with Example

To illustrate the general idea behind top-Down parsing, we begin with a vague description of the algorithm, tune it for efficiency, and then formalize it to obtain our LL(1) algorithm. Since this is top-down parsing, we must begin at our start symbol S' and choose a rule to apply. Given that we choose to work with augmented grammars, there is a guarantee that there will be only one rule. We apply the rule to get our first α, i.e., a step in our derivation; α_1 is $\vdash$ S $\dashv$. This was our setup stage. At this point, we apply the following loop:

From left to right, look at the symbols in the current α_i. If the symbol is a terminal, match it to the terminal at the same position in the input string s. If it does not match, report a parsing error and terminate (if we have a terminal, that's part of the actual string, so it really does have to match!). Otherwise, continue matching terminals until we hit the first non-terminal in α_i. This is the leftmost non-terminal. Choose an appropriate production rule for this non-terminal and apply it, i.e., replace the non-terminal with the right-hand side of the chosen rule. This gives α_{i+1}. Repeat the algorithm until we either generate a parse error or no more non-terminals are left.

Let's look at an example parse using the following augmented grammar (we just list the production rules; upper case letters are non-terminals and lower-case letters are terminals):

S' $\rightarrow$ $\vdash$ S $\dashv$
S $\rightarrow$ AyB
A $\rightarrow$ ab
A $\rightarrow$ cd
B $\rightarrow$ z
B $\rightarrow$ wx

Suppose our input string s is $\vdash$ a b y w x $\dashv$. At the setup stage, we get α_1 as $\vdash$ S $\dashv$.

- We begin looking at symbols in α_1 left to right. The first symbol is $\vdash$ which matches the first symbol in the string s. We move to the next symbol and see that it is the non-terminal S. We apply the rule $S \rightarrow AyB$ to obtain α_2, $\vdash$ AyB $\dashv$.

- We begin the process again; $\vdash$ from α_2 matches the first symbol in the string s and then we encounter the leftmost nonterminal, A. The algorithm chooses the appropriate rule by sneaking a peek at what the **next unmatched** symbol in the input is (it is a). The algorithm chooses to apply $A \rightarrow ab$. This gives us α_3, $\vdash$ abyB $\dashv$.

- The process repeats: the algorithm successively matches $\vdash$, then a, then b, then y with the input and then hits the leftmost non-terminal B. By sneaking a peek at the next unmatched symbol (it is w) the algorithm chooses the appropriate rule $B \rightarrow wx$ to produce α_4, $\vdash$ abywx $\dashv$.

- The process repeats once again: the algorithm matches $\vdash$, then a, then b, then y, then w, then x and then $\dashv$. At this point, there are no symbols left in α_4 and there are no unmatched symbols in the input string. The parse was successful.

The derivation obtained : $S' \Rightarrow \underbrace{\vdash S \dashv}_{\alpha_1} \Rightarrow \underbrace{\vdash AyB \dashv}_{\alpha_2} \Rightarrow \underbrace{\vdash abyB \dashv}_{\alpha_3} \Rightarrow \underbrace{\vdash abywx \dashv}_{\alpha_4} = s$

2.1 Observation 1: What is an "Appropriate" Rule

As you followed along the example above, you likely noticed the use of the term "appropriate rule". We also talked about peeking at the next input symbol and then magically knowing which rule to pick. "Sneaking a peek" has a more formal term; it is called using a *lookahead*. We used a lookahead of 1 (looked at 1 next symbol from the input). How does looking ahead help? We will soon discuss how we can create a table `Predict[A][a]` that, given a non-terminal A and a lookahead terminal a, will predict which rule to choose for A.

2.2 Observation 2: This is Inefficient

As you followed along with the algorithm on the input string, you must have noticed that we are redoing some steps again and again, matching already matched terminals from α_i to the input string once again while reading through α_{i+1}. Formally, each α_i is some string of terminals and non-terminals. Assuming that there is at least one non-terminal, i.e., we haven't derived a string of terminals just yet, we can write α_i as $xA\beta$ where x is a string of terminals, A is the leftmost non-terminal and β is some, possibly empty, string of terminals and non-terminals. While processing α_i, we match all of x to the input. Then we apply a rule for A, say $A \to \gamma$. This makes α_{i+1} be $x\gamma\beta$. The process repeats and the first thing we do is rematch all of x. This is redundant.

An obvious optimization therefore is to stop tracking the prefix of the input (and α_i) that has already been matched. Looking back, when α_i was $xA\beta$ and we match x to the input, we will only keep track of the remaining $A\beta$. Then we apply the rule for A and this will produce $\gamma\beta$. Next, we will look at the first symbol in $\gamma\beta$ and continue with our algorithm.

Taking this one step further, a practical thought is how to keep track of this truncated part of α_i. What if we stored the symbols on a stack with the leftmost symbol on the top of the stack? Going back to the example we traced, since we always begin with α_1 as $\vdash$ S $\dashv$, we can begin with the stack contents such that $\vdash$ is on the top of the stack and $\dashv$ at the bottom. The advantage will be visible if we run through the example again (which we do in the next section). Since drawing stacks vertically isn't very compact, we will flatten our stacks to the side with the top of the stack (TOS) appearing first (on the left), e.g., $\vdash$ S $\dashv$. We choose this format since this is exactly how we have been representing our non-stack versions of α_i.

2.3 The Top-Down Parsing Algorithm

We can reword our algorithm based on our observations and decisions. Start by pushing the start symbol of the grammar on the stack. While the TOS is a terminal, pop it and match it against input. If it does not match, report a parse error. Otherwise continue with the next symbol. If TOS is a non-terminal A, pop it and query `Predict[A][a]` where a is the first unmatched symbol from the input. If `Predict[A][a]` has no rule for us, reject with a parse error. Otherwise, if `Predict[A][a]` returns a rule $A \to \gamma$, apply the rule by pushing the symbols in γ in reverse. We push the symbols in reverse so that the leftmost symbol ends up on TOS, and therefore, reading from the top of the stack to the bottom gives the symbols of γ in the right order. This also ensures that the first non-terminal we encounter is the leftmost non-terminal (recall we are aiming for leftmost derivations). Repeat until we get a parse error or the stack is empty, at which point we accept. At any given point, the stack contents represent the truncated α_i, that is, the current part of the derivation with the matched prefix removed.

This can be formalized into the following algorithm:

Algorithm 1 Top-Down Parsing Algorithm

```
1:  push S'
2:  for each 'a' in ⊢ input ⊣ do
3:      while top of stack is A ∈ N do
4:          pop A
5:          if Predict[A][a] gives A →γ then
6:              push the symbols in γ (right to left)
7:          else
8:              reject
9:          end if
10:     end while
11:     // TOS is a terminal
12:     if TOS is not 'a' then
13:         Reject
14:     else
15:         pop 'a'
16:     end if
17: end for
18: Accept // stack is necessarily empty
```

We trace through the same grammar and input string using the algorithm just discussed. The Predict table is shown. The numbers in some cells of the Predict table specify which rule to apply given a non-terminal and lookahead, while an empty cell means there is no appropriate rule.

(1) $S' \rightarrow \vdash$ S $\dashv$
(2) S $\rightarrow$ AyB
(3) A $\rightarrow$ ab
(4) A $\rightarrow$ cd
(5) B $\rightarrow$ z
(6) B $\rightarrow$ wx

	$\vdash$	a	b	c	d	w	x	y	z	$\dashv$
S'	1									
S		2		2						
A		3		4						
B						6			5	

Input string s is $\vdash$ a b y w x $\dashv$.

We have created a video that traces the Top-Down Parsing Algorithm using an alternate grammar.

An interesting thing to note is that at any time during the algorithm, we can obtain α_i by concatenating the contents in the `Read` and `Stack` columns.

Recall that the goal is to obtain a derivation. The rules chosen during the execution of the algorithm give the derivation, i.e., Rule 1 followed by Rule 2, then Rule 3 and finally Rule 6 (all applied to the leftmost non-terminal): $S' \Rightarrow \vdash$ S $\dashv \Rightarrow \vdash$ AyB $\dashv \Rightarrow \vdash$ abyB $\dashv \Rightarrow \vdash$ abywx $\dashv$. This uniquely defines a parse tree as discussed in the previous module.

The algorithm can produce an error in one of two ways:

1. The TOS is a terminal, but it does not match the next input symbol.
2. The algorithm queries Predict[A][a] for some A and a and finds either no rule, or more than one rule.

Read	Unread	Stack	Action
ϵ	$\vdash$ a b y w x $\dashv$	S'	pop S', Predict[S'][$\vdash$] gives rule 1 push $\dashv$, S, $\vdash$
ϵ	$\vdash$ a b y w x $\dashv$	$\vdash$ S $\dashv$	Match $\vdash$
$\vdash$	a b y w x $\dashv$	S $\dashv$	pop S, Predict[S][a] gives rule 2 push B, y, A
$\vdash$	a b y w x $\dashv$	A y B $\dashv$	pop A, Predict[A][a] gives rule 3 push b, a
$\vdash$	a b y w x $\dashv$	a b y B $\dashv$	match a
$\vdash$ a	b y w x $\dashv$	b y B $\dashv$	match b
$\vdash$ a b	y w x $\dashv$	y B $\dashv$	match y
$\vdash$ a b y	w x $\dashv$	B $\dashv$	pop B, Predict[B][w] gives rule 6 push x, w
$\vdash$ a b y	w x $\dashv$	w x $\dashv$	match w
$\vdash$ a b y w	x $\dashv$	x $\dashv$	match x
$\vdash$ a b y w x	$\dashv$	$\dashv$	match $\dashv$
$\vdash$ a b y w x$\dashv$	ϵ	ϵ	Accept

The algorithm we just discussed is called LL(1) parsing. The first L represents the Left-to-Right scan of input, the second L indicates that the algorithm produces Leftmost derivations[1], and the 1 indicates that we looked ahead at 1 symbol.

Definition 1 *A grammar is **LL(1)** if and only if each cell of the Predict table contains at most one rule.*

The only thing left for us to do is to determine how to create the Predict table.

3 Constructing the Predict Table

We have created the Building a Predict Table video that provides the intuition of how the predict table is computed. We recommend watching the video before continuing below.

We have talked about the Predict table as a two-dimensional lookup table. But really, it is a function that produces a set of rules that can apply when $A \in N'$ (a non-terminal) is on the TOS and $a \in T'$ (a terminal) is the next input symbol. We will iteratively refine our definition of this function. As a first step, we can include rules of the form A $\rightarrow \beta$ if $\beta \Rightarrow^* a\gamma$, i.e., if β can derive a string where the *first* symbol is an a then we could apply the rule A $\rightarrow \beta$ since we will be able to get to a. Formally,

$\text{First}(\beta) = \{a \in T' : \beta \Rightarrow^* a\gamma, \text{ for some } \gamma \in V^*\}$
i.e., the set of terminals that can be the first symbol in a string derived from $\beta \in V^*$.

[1]Most parsers parse left to right simply because it's confusing to detect parse errors at the end first. With LL, it's actually the fact that it's top-down that forces a left-to-right scan to also produce a leftmost derivation. If we scanned right-to-left using the same algorithm, we would have RR(1) parsing. When we get to bottom-up parsing, we'll see that it's called LR parsing, and similarly, the fact that it gets a rightmost derivation is an inevitable part of parsing bottom-up, left-to-right; bottom-up parsing done right-to-left would be RL parsing.

Predict[A][a] = {A $\rightarrow \beta$: $a \in$ First(β) }

The Predict function as described above misses some rules. To understand this, consider the following grammar and its Predict table based on our current definition of this function:

(1) $S' \rightarrow$ XY
(2) X $\rightarrow \epsilon$
(3) Y $\rightarrow$ z

	z
S'	1
X	
Y	3

Notice that $S' \Rightarrow$ XY $\Rightarrow$ Y $\Rightarrow$ z is a derivation for the string z. However, if we use our algorithm, we will get a parser error: Push S'. Then, pop S'. Querying Predict[S'][z] gives us rule 1 so we push Y then X on to the stack. The algorithm then queries Predict[X][z] and there is no rule for this. We haven't made a mistake in this table entry since for the only rule for X, X $\rightarrow \epsilon$, First(ϵ) = {} and therefore z $\notin$ First(ϵ).

What is missing from our definition of Predict is accounting for rules where the non-terminal is nullable, i.e., it derives ϵ. While computing Predict[A][a], a rule of the form A $\rightarrow \epsilon$ might make sense as perhaps a does not need to be derived from A but from some symbol that follows A. We define the following:

Nullable(β) = true iff $\beta \Rightarrow^* \epsilon$ and false otherwise, i.e., Nullable(β) is true only if β can derive the empty string.

Follow(A) = $\{b \in T' : S' \Rightarrow^* \alpha A b \beta$ for some $\alpha, \beta \in V^*\}$, i.e., Follow(A) is a set of terminals that can come immediately after A in a derivation starting at the start symbol S'.

Based on this, we can update our definition for Predict to this correct definition:

$$\text{Predict}[A][a] = \{A \rightarrow \beta : a \in \text{First}(\beta)\} \cup \{A \rightarrow \beta : \beta \text{ is nullable and } a \in \text{Follow}(A)\}$$

For our small example from above, while computing Predict[X][z], the rule for X is X $\rightarrow \epsilon$. Since ϵ by definition is nullable, we find the follow set of X (we will see an algorithm to compute this later). Follow(X) should contain z since $S' \Rightarrow$ XY $\Rightarrow$ Xz, i.e., we can derive a string where X is followed by z (notice that the derivation style is unimportant in the computation of the Follow set). This means, Predict[X][z] contains X $\rightarrow \epsilon$. Now our parse will succeed!

While the definitions of First, Follow and Nullable are enough to compute the Predict function, we give algorithms to do this programmatically.

3.1 Computing Nullable

We defined Nullable as, Nullable(β) = true iff $\beta \Rightarrow^* \epsilon$ and false otherwise.

We begin with some observations:

- Nullable(ϵ) = true by definition
- Nullable(β) = false whenever β contains a terminal symbol.

- Nullable(AB) = Nullable(A) $\wedge$ Nullable(B)

Based on these observations, it suffices to compute Nullable(A) for all $A \in N'$.

Algorithm 2 Nullable(A) for all $A \in N'$

```
1: Initialize Nullable(A) = false for all A ∈ N'.
2: repeat
3:     for each production in P do
4:         if (P is A → ϵ) or (P is A → B_1 ··· B_k and ⋀_{i=1}^{k} Nullable(B_i) = true) then
5:             Nullable(A) = true
6:         end if
7:     end for
8: until nothing changes
```

A non-terminal A is nullable if it directly derives ϵ or has a rule of the form $A \rightarrow B_1 \cdots B_k$ where each of B_1 through B_k are nullable. Notice that the identification of a new nullable non-terminal may lead to determining other non-terminals to be nullable. Therefore, the algorithm must continue until a complete pass through all rules does not identify any new nullable non-terminal.

(1) $S' \rightarrow \vdash$ S $\dashv$
(2) S $\rightarrow$ b S d
(3) S $\rightarrow$ p S q
(4) S $\rightarrow$ C
(5) C $\rightarrow$ l C
(6) C $\rightarrow \epsilon$

Iteration	0	1	2	3
S'	false	false	false	false
S	false	false	true	true
C	false	true	true	true

When the algorithm starts (iteration 0), all non-terminals are marked as non-nullable. On the first iteration, C is marked nullable due to Rule 6. On the second iteration, S is marked nullable due to Rule 4 and the fact that C is nullable. On the third iteration, no new non-terminals are marked as nullable, so we are done.

3.2 Computing First

We defined First as, First(β) = $\{a \in T' : \beta \Rightarrow^* a\gamma, \text{ for some } \gamma \in V^*\}$.

We begin by presenting an algorithm for computing First(A) where A is a non-terminal.

Algorithm 3 First(A) for all $A \in N'$

```
1: Initialize First(A) = {} for all A ∈ N'.
2: repeat
3:     for each rule A → B_1 B_2 ··· B_k in P do
4:         for i ∈ {1, .., k} do
5:             if B_i ∈ T' then
6:                 First(A) = First(A) ∪ {B_i}; break
7:             else
8:                 First(A) = First(A) ∪ First(B_i)
9:                 if Nullable(B_i) == False then break
10:            end if
11:        end for
12:    end for
13: until nothing changes
```

The algorithm inspects each rule of the form A $\rightarrow \beta$. Let β be $B_1 B_2 \cdots B_k$. The algorithm begins to sequentially assess each B_i beginning at B_1. If B_i is a terminal, the terminal is added to First(A) and

the rest of β is not processed since it can no longer contribute to First(A) (as a terminal has already been encountered). If B_i happens to be a non-terminal, then First(A) should contain First(B_i). Additionally, we only look at B_{i+1} if B_i is nullable (as otherwise B_{i+1} cannot contribute to First(A)).

The above algorithm computes the First set for each non-terminal in the language. It is also useful to have the ability to compute the First set of an arbitrary β, i.e., not necessarily the right-hand side of a rule. We will, for example, use this extension in the computation of the Follow sets.

Algorithm 4 First(β) where $\beta = B_1 \cdots B_n \in V^*$

```
result = ∅
for i ∈ {1, .., n} do
    if B_i ∈ T' then
        result = result ∪ {B_i}; break
    else
        result = result ∪ First(B_i)
        if Nullable(B_i) == False then break
    end if
end for
```

Notice that First(A) can actually be written in a way that it uses First(β) for the right-hand side of each rule it assesses. We, on purpose, duplicate the algorithm for readability.

(1) $S' \rightarrow \vdash$ S $\dashv$
(2) S $\rightarrow$ b S d
(3) S $\rightarrow$ p S q
(4) S $\rightarrow$ C
(5) C $\rightarrow$ l C
(6) C $\rightarrow \epsilon$

Iteration	0	1	2	3
S'	{}	{$\vdash$}	{$\vdash$}	{$\vdash$}
S	{}	{b,p}	{b,p,l}	{b,p,l}
C	{}	{l}	{l}	{l}

We begin (iteration 0) with empty sets for all the non-terminals in the grammar. On the first iteration, $\vdash$ is added to First(S') since in Rule 1, the first symbol on the right-hand side of the rule is $\vdash$. Similarly, b and p are added to First(S) due to Rules 2 and 3, respectively. First(C) is updated by adding l to it due to Rule 5. In iteration 2, the only change is to First(S) due to Rule 4. The right-hand side of Rule 4 is the non-terminal C. So, we add to First(S) the symbols in First(C), which is the symbol l. On the third iteration, no First set changes, so we are done.

3.3 Computing Follow

We defined Follow as, Follow(A) = $\{b \in T' : S' \Rightarrow^* \alpha A b \beta$ for some $\alpha, \beta \in V^*\}$.

Algorithm 5 Follow(A) for all $A \in N$ (Recall N = $N' \setminus \{ S' \}$)

```
Initialize Follow(A) = {} for all A ∈ N.
repeat
    for each production A → B_1 B_2 ··· B_k in P' do
        for i ∈ {1, .., k} do
            if B_i ∈ N then
                Follow(B_i) = Follow(B_i) ∪ First(B_{i+1} ··· B_k)
                if ⋀_{m=i+1}^{k} Nullable(B_m) == True or i == k then
                    Follow(B_i) = Follow(B_i) ∪ Follow(A)
                end if
            end if
        end for
    end for
until nothing changes
```

The algorithm goes through each rule $A \to B_1 \cdots B_k$ and updates Follow(B_i) to contain First($B_{i+1} \cdots B_k$) since First($B_{i+1} \cdots B_k$) is the set of terminals that can occur as the first symbol in a derivation starting with $B_{i+1} \cdots B_k$, which means they can follow B_i. Additionally, Follow(B_i) should also contain the Follow(A) if B_i is the last symbol in the rule or if all symbols after B_i are nullable.

(1) $S' \to \vdash$ S $\dashv$
(2) S $\to$ b S d
(3) S $\to$ p S q
(4) S $\to$ C
(5) C $\to$ l C
(6) C $\to \epsilon$

Iteration	0	1	2
S	{}	{⊣,d,q}	{⊣,d,q}
C	{}	{⊣,d,q}	{⊣,d,q}

As for the previous algorithms, we initialize our sets to empty sets in iteration 0. On the first iteration, the symbols ⊣, d and q are added to Follow(S) since, on the right-hand side of Rules 1, 2 and 3 respectively, S is followed by these three symbols. To see how Follow(C) is updated, we look at Rule 4. Since C is not followed by any symbol on the right-hand side of this rule, we must add Follow(S) to Follow(C). We just updated Follow(S) to contain ⊣, d and q. These get added to Follow(C). Notice that had we not already computed Follow(S), Follow(S) would have been an empty set and we would have required an extra iteration to update Follow(C). The next iteration does not change any of the sets, so the algorithm stops.

3.4 Computing the Predict Table

We defined Predict as:
Predict[A][a] = {A $\to \beta$: a $\in$ First(β)} $\cup$ { A $\to \beta$: β is nullable and $a \in$ Follow(A) }

In the previous sections, we looked at algorithms that compute the individual components. We now look at the algorithm that computes the Predict table leveraging these previously described algorithms.

Algorithm 6 Computing Predict

```
1: Initialize Predict[A][a] = {} for all A ∈ N' and a ∈ T'.
2: for each production A → β in P do
3:     for each a ∈ First(β) do
4:         Add A → β to Predict[A][a]
5:     end for
6:     if Nullable(β) == True then
7:         for each a ∈ Follow(A) do
8:             Add A → β to Predict[A][a]
9:         end for
10:    end if
11: end for
```

The algorithm begins by initializing each Predict[A][a] to the empty set of rules. Unlike the previous algorithms, computing the Predict table does not require multiple iterations. The algorithm goes through each rule in the grammar. While processing a grammar rule of the form $A \to \beta$, the algorithm updates the row for A in the Predict Table. First, we use the algorithm for computing First for an arbitrary right-hand side and compute First(β). Then, Predict[A][a] is updated to contain A $\to \beta$ for each symbol a $\in$ First(β). For the same rule, A $\to \beta$, we compute Nullable(β) using the Nullable values for non-terminals. If Nullable(β) is true, we add A $\to \beta$ to Predict[A][a]

for each a ∈ Follow(A).

We have summarized the nullable predicate and the First and Follow sets below for convenience.

(1) $S' \rightarrow \vdash$ S $\dashv$
(2) S → b S d
(3) S → p S q
(4) S → C
(5) C → l C
(6) C → ϵ

Summary:

	Nullable	First	Follow
S'	false	{⊢}	{}
S	true	{b,p,l}	{⊣,d,q}
C	true	{l}	{⊣,d,q}

Predict Table:

	⊢	⊣	b	d	p	q	l
S'	1						
S		4	2	4	3	4	4
C		6		6		6	5

To see how the algorithm computes the Predict Table (shown above to the right), let's consider each rule in the grammar:

1. $S' \rightarrow \vdash$ S $\dashv$. First(⊢ S ⊣) is {⊢}, therefore Rule 1 is added to Predict[S'][⊢]. Since Nullable(⊢ S ⊣) is false, the rule is not added to any other entry in the row for S'.
2. S → b S d. First(b S d) is {b}, therefore Rule 2 is added to Predict[S][b]. Since Nullable(b S d) is false, the rule is not added to any other entry in the row for S.
3. S → p S q. Like the above, this causes Rule 3 to be added to Predict[S][p].
4. S → C. First(C) is {l} which means Rule 4 is added to Predict[S][l]. Also, Nullable(C) is true. This means Rule 4 is added to all entries in the row for S for the terminals in Follow(S), {⊣,d,q}.
5. C → l C. First(l C) is {l}. So, Rule 5 is added to Predict[C][l].
6. C → ϵ. First(ϵ) is {} so the rule is not added to any entry in the row for C. But Nullable(ϵ) is true by definition. Therefore, Rule 6 is added to every column in the row for C where the terminal is in Follow(C), {⊣,d,q}.

Recall our definition of LL(1). Looking at the Predict table, we see that each entry in the Predict Table is at most one rule. This grammar is LL(1).

Trace the LL(1) parsing algorithm using the input string ⊢ b p l q d ⊣ using the grammar and predict table just constructed. Your steps should include columns for `Read`, `Unread`, `Stack` and `Action` as shown earlier in this module (Section 2.3).

[2]

Trace the LL(1) parsing algorithm using the input string ⊢ b l b d ⊣ using the grammar and predict table just constructed. Your steps should include columns for `Read`, `Unread`, `Stack` and `Action` as shown earlier in this module (Section 2.3).

[3]

[2] Answer can be found in Section 6.1
[3] Answer can be found in Section 6.2

Using the algorithms discussed above, create tables showing the iterations in computing the nullable, First and Follow sets for the following grammar:

(0) $S' \rightarrow$ $\vdash$ S $\dashv$
(1) S $\rightarrow$ c
(2) S $\rightarrow$ QRS
(3) Q $\rightarrow$ R
(4) Q $\rightarrow$ d
(5) R $\rightarrow$ ϵ
(6) R $\rightarrow$ b

[4]

For the grammar below (same as the grammar above) compute the Predict Table. The values for nullable, First and Follow are provided:

(0) $S' \rightarrow$ $\vdash$ S $\dashv$
(1) S $\rightarrow$ c
(2) S $\rightarrow$ QRS
(3) Q $\rightarrow$ R
(4) Q $\rightarrow$ d
(5) R $\rightarrow$ ϵ
(6) R $\rightarrow$ b

	Nullable	First	Follow
S'	False	$\{\vdash\}$	$\{\}$
S	False	$\{b, c, d\}$	$\{\dashv\}$
Q	True	$\{b, d\}$	$\{b, c, d\}$
R	True	$\{b\}$	$\{b, c, d\}$

[5]

Construct the four tables (Nullable, First, Follow and Predict) for the following grammar:

(0) $S' \rightarrow$ $\vdash$ S $\dashv$
(1) S $\rightarrow$ Bb
(2) S $\rightarrow$ Cd
(3) B $\rightarrow$ aB
(4) B $\rightarrow$ ϵ
(5) C $\rightarrow$ cC
(6) C $\rightarrow$ ϵ

[6]

Given the following grammar (from above) and Predict Table, trace the LL(1) parsing algorithm using the input strings $\vdash$ a b $\dashv$ and $\vdash$ a b c $\dashv$:

(0) $S' \rightarrow$ $\vdash$ S $\dashv$
(1) S $\rightarrow$ Bb
(2) S $\rightarrow$ Cd
(3) B $\rightarrow$ aB
(4) B $\rightarrow$ ϵ
(5) C $\rightarrow$ cC
(6) C $\rightarrow$ ϵ

Predict:

	$\vdash$	a	b	c	d	$\dashv$
S'	0					
S		1	1	2	2	
B		3	4			
C				5	6	

[7]

[4] Answer can be found in Section 6.3
[5] Answer can be found in Section 6.4
[6] Answer can be found in Section 6.5
[7] Answer can be found in Section 6.6

4 Parse Trees

Recall that we have been discussing top-down parsing as a way of obtaining a leftmost derivation for an input string with the eventual goal of generating the parse tree that represents the structure of the input. Consider the following grammar and predict table. (The Nullable, First and Follow sets are given for completeness):

(0) S' → ⊢ S ⊣
(1) S → T Z
(2) Z → + T Z
(3) Z → ϵ
(4) T → F T'
(5) T' → ∗ F T'
(6) T' → ϵ
(7) F → a
(8) F → b
(9) F → c

	Nullable	First	Follow
S'	False	$\{\vdash\}$	$\{\}$
S	False	$\{a, b, c\}$	$\{\dashv\}$
Z	True	$\{+\}$	$\{\dashv\}$
T	False	$\{a, b, c\}$	$\{\dashv, +\}$
T'	True	$\{*\}$	$\{\dashv, +\}$
F	False	$\{a, b, c\}$	$\{\dashv, +, *\}$

Predict Table:

	⊢	a	b	c	+	∗	⊣
S'	0						
S		1	1	1			
Z					2		3
T		4	4	4			
T'					6	5	6
F		7	8	9			

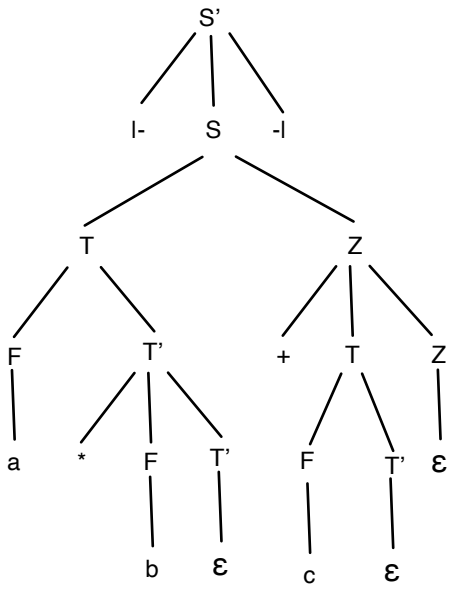
Let's prove that the string ⊢ a ∗ b + c ⊣ is in the language by providing a derivation for it:

Read	Unread	Stack	Action
ϵ	⊢ a ∗ b + c ⊣	S'	pop S', Predict[S'][⊢] gives rule 0, push ⊣, S, ⊢
ϵ	⊢ a ∗ b + c ⊣	⊢ S ⊣	Match ⊢
⊢	a ∗ b + c ⊣	S ⊣	pop S, Predict[S][a] gives rule 1, push Z, T
⊢	a ∗ b + c ⊣	T Z ⊣	pop T, Predict[T][a] gives rule 4, push T', F
⊢	a ∗ b + c ⊣	F T' Z ⊣	pop F, Predict[F][a] gives rule 7, push a
⊢	a ∗ b + c ⊣	a T' Z ⊣	match a
⊢ a	∗ b + c ⊣	T' Z ⊣	pop T', Predict[T'][∗] gives rule 5, push T', F, ∗
⊢ a	∗ b + c ⊣	∗ F T' Z ⊣	match ∗
⊢ a ∗	b + c ⊣	F T' Z ⊣	pop F, Predict[F][b] gives rule 8, push b
⊢ a ∗	b + c ⊣	b T' Z ⊣	match b
⊢ a ∗ b	+ c ⊣	T' Z ⊣	pop T', Predict[T'][+] gives rule 6
⊢ a ∗ b	+ c ⊣	Z ⊣	pop Z, Predict[Z][+] gives rule 2, push Z, T, +
⊢ a ∗ b	+ c ⊣	+ T Z ⊣	match +
⊢ a ∗ b +	c ⊣	T Z ⊣	pop T, Predict[T][c] gives rule 4, push T', F
⊢ a ∗ b +	c ⊣	F T' Z ⊣	pop F, Predict[F][c] gives rule 9, push c
⊢ a ∗ b +	c ⊣	c T' Z ⊣	match c
⊢ a ∗ b + c	⊣	T' Z ⊣	pop T', Predict[T'][⊣] gives rule 6
⊢ a ∗ b + c	⊣	Z ⊣	pop Z, Predict[Z][⊣] gives rule 3
⊢ a ∗ b + c	⊣	⊣	match ⊣
⊢ a ∗ b + c ⊣	ϵ	ϵ	Accept

Since the top-down parsing algorithm accepts the given input, we know that the input is a word in the language specified by the grammar. The derivation is the sequence of rules that were applied (Rules 0, 1, 4, 7, 5, 8, 6, 2, 4, 9, 6 and 3 in that order). Recall also from our discussion of the algorithm that the concatenation of the `Read` input and the `Stack` contents always gives us the α_is

obtained during the derivation. A parser can use the derivation to generate the parse tree. We show the derivation and the corresponding parse tree below:

$$
\begin{array}{lll}
S' \Rightarrow \vdash \text{S} \dashv & & (\text{rule } 0) \\
\Rightarrow \vdash \text{T Z} \dashv & & (\text{rule } 1) \\
\Rightarrow \vdash \text{F } T' \text{ Z} \dashv & & (\text{rule } 4) \\
\Rightarrow \vdash \text{a } T' \text{ Z} \dashv & & (\text{rule } 7) \\
\Rightarrow \vdash \text{a} * \text{F } T' \text{ Z} \dashv & & (\text{rule } 5) \\
\Rightarrow \vdash \text{a} * \text{b } T' \text{ Z} \dashv & & (\text{rule } 8) \\
\Rightarrow \vdash \text{a} * \text{b Z} \dashv & & (\text{rule } 6) \\
\Rightarrow \vdash \text{a} * \text{b} + \text{T Z} \dashv & & (\text{rule } 2) \\
\Rightarrow \vdash \text{a} * \text{b} + \text{F } T' \text{ Z} \dashv & & (\text{rule } 4) \\
\Rightarrow \vdash \text{a} * \text{b} + \text{c } T' \text{ Z} \dashv & & (\text{rule } 9) \\
\Rightarrow \vdash \text{a} * \text{b} + \text{c Z} \dashv & & (\text{rule } 6) \\
\Rightarrow \vdash \text{a} * \text{b} + \text{c} \dashv & & (\text{rule } 3)
\end{array}
$$



The Building a Parse Tree video discusses the way parse trees are constructed by Top Down Parsers.

5 Limitations of LL(1) Grammars

We defined a grammar to be LL(1) if and only if each cell of the predict table contains at most one rule. Consider the following simple grammar for addition of expressions (Note that contrary to previous examples, there is no reason why terminal/non-terminal "symbols" have to be one character).

(1) expr → expr op expr
(2) expr → ID
(3) op → +

Instead of creating the predict table (you certainly could), we give reasoning to show that this grammar is not LL(1). Consider the set for Predict[`expr`][`ID`]:

- We consider whether Rule 1 should be in Predict[`expr`][`ID`]. We find the First set for the right-hand side of this rule. We know that First(`expr`) ⊆ First(`expr op expr`). We know from `expr` → `ID` that `ID` ∈ First(`expr`). Therefore ID ∈ First(`expr op expr`). Therefore, Rule 1 ∈ Predict[`expr`][`ID`].
- We consider whether Rule 2 should be in Predict[`expr`][`ID`]. By definition, First(`ID`) = {`ID`} since `ID` is a terminal. Therefore, Rule 2 ∈ Predict[`expr`][`ID`].

Since Predict[`expr`][`ID`] contains two rules, we conclude that the grammar is **not** LL(1).

In fact, reasoning, based on the definitions of Nullable, First and Follow, we can conclude that a grammar is $LL(1)$ if and only if:

- no two distinct rules with the same LHS can generate the same first terminal

- no nullable symbol A has the same terminal a in both its first and follow sets
- there is only one way to derive ϵ from a nullable symbol

The following grammar is not LL(1). Give reasoning to prove this claim.

```
    (1) S → S + T
    (2) S → T
    (3) T → T * F
    (4) T → F
(5,6,7) F → a | b | c
```

[8]

In case you did not realize, this was the grammar we constructed in the previous Module to respect BEDMAS (giving higher precedence to multiplication over addition by making it deeper in the tree). We have just shown that the grammar is not LL(1), i.e., we cannot use one lookahead in our top-down parsing algorithm to derive input strings. The primary issue is that left recursion is at odds with $LL(1)$. In fact, left recursive grammars are **never** LL(1). To recall, a left recursive grammar, as the name indicates, is a grammar where the recursion on a non-terminal happens on the left within the right-hand side. For example, the rule S → S + T is left recursive, the S symbol within S + T is on the left. A Right Recursive rule would take the form S → T + S.

If we want to have success with parsing this grammar, we must at least make it right recursive.

5.1 Writing Right Recursive Grammars

We can simply switch the recursion to the right. This would change the parse tree, and thus the meaning, of any input string, so it's not a good solution, but let's at least explore whether it works:

```
    (1) S → T + S
    (2) S → T
    (3) T → F * T
    (4) T → F
(5,6,7) F → a | b | c
```

Is the grammar presented above LL(1)?

[9]

The problem is that given S on the stack and the lookahead of just a, the algorithm cannot decide whether it should use S → T (since it does not know what comes after the a) or it should use S → T + S.

The problem here is that the left-hand side of the two rules for S have a common prefix (T).

[8] This grammar is not LL(1) because we have two distinct rules with the same LHS that can generate the same terminal. More formally, we can show that S → T ∈ Predict[S][a] since { a } ⊆ First(F) ⊆ First(T) . Similarly, we can show that S → S + T ∈ Predict[S][a] since { a } ⊆ First(F) ⊆ First(T) ⊆ First(S) ⊆ First(S+T).

[9] Sadly, no! The grammar is still not LL(1). Predict[S][a] contains both rules for S.

A grammar with two or more rules for the same non-terminal with a common left prefix of length k, cannot be LL(k).

In our example, the common left prefix is of length 1 and therefore the grammar is not LL(1). As discussed, one solution is to increase the lookahead. The grammar above is LL(2). Note that we don't provide an algorithm for generating the predict table for an LL(2) parser, because LL(2) parsing is very rarely used in practice and generating such predict tables is much more complex than for LL(1). If we want to continue with an LL(1) algorithm, we can try left factorization.

Suppose a grammar has rules $A \to \alpha\beta_1|...|\alpha\beta_n$, all with a common prefix of $\alpha \neq \epsilon$ on the right-hand side. Then, we can change these to the following equivalent grammar by **left factoring**:

A –> αB
B –> β_1| . . . | β_n

Applying this technique to the previous example:

Left Recursive:	Right recursive:	Right recursive factored:
(1) S → S + T	(1) S → T + S	(1) S → T X
(2) S → T	(2) S → T	(2,3) X → + S | ϵ
(3) T → T ∗ F	(3) T → F ∗ T	(4) T → F Y
(4) T → F	(4) T → F	(5,6) Y → ∗ T | ϵ
(5,6,7) F → a|b|c	(5,6,7) F → a|b|c	(7,8,9) F → a|b|c

Notice that the right-recursive and factored grammar is $LL(1)$ as seen in the Predict table below.

	Nullable	First	Follow
S	false	{a,b,c}	{}
X	true	{+}	{}
T	false	{a,b,c}	{+}
Y	true	{∗}	{+}
F	false	{a,b,c}	{+,∗}

Predict Table:

	a	b	c	+	∗
S	1	1	1		
X				2	
T	4	4	4		
Y				6	5
F	7	8	9		

An alternate approach to converting a left-recursive grammar to be right recursive is to apply a different transformation: replace each pair of rules A → Aα and A → β where β does not begin with the non-terminal A with the rules:

A –> βA'
A' –> αA' | ϵ

Using this on our grammar we get:

Left Recursive:	Right Recursive:
(1) S → S + T	(1) S –> T Z'
(2) S → T	(2,3) Z' –> + T Z' | ϵ
(3) T → T ∗ F	(4) T –> F T'
(4) T → F	(5,6) T' –> ∗ F T' |ϵ
(5,6,7) F → a|b|c	(7,8,9) F –> a|b|c

Notice that this transformation produces a grammar almost the same as the one from above. This grammar is also LL(1) with the same Predict table (barring the change to non-terminal names). However, there is a problem with both of our right recursive LL(1) grammars: they are right associative. While this does not pose an immediate problem in this grammar (multiplication has higher precedence than addition and there is no subtraction and division), consider that in general we would also have other operators at the same precedence level. For example, imagine that we have subtraction at the same precedence level as addition. Since this grammar is right associative, the expression x-y+z would become subtract the result of adding y and z from x rather than what we are commonly used to, subtract y from x and then add z (left associativity). The right associative grammar would require the programmer to introduce parentheses to force the left association, e.g. (x-y)+z.

The Fixing Factored Grammars video discusses how Top Down Parsers deal with transformed grammars.

The Top-Down parsing algorithm we have studied, $LL(1)$, is not compatible with left-associative grammars. We discuss an alternate, even more powerful, parsing algorithm in the next Module.

6 Answers to longer self-practice questions

6.1 Self-practice answer 1

(1) $S' \rightarrow \vdash$ S $\dashv$
(2) S $\rightarrow$ b S d
(3) S $\rightarrow$ p S q
(4) S $\rightarrow$ C
(5) C $\rightarrow$ lC
(6) C $\rightarrow \epsilon$

Predict Table:

	$\vdash$	$\dashv$	b	d	p	q	l
S'	1						
S		4	2	4	3	4	4
C		6		6		6	5

Trace the LL(1) parsing algorithm using the input string $\vdash$ b p l q d $\dashv$ using the grammar and predict table just constructed. Your steps should include columns for `Read`, `Unread`, `Stack` and `Action` as shown earlier in this module (Section 2.3).

Read	Unread	Stack	Action
ϵ	$\vdash$ b p l q d $\dashv$	S'	pop S', Predict[S'][$\vdash$] gives rule 1 push $\dashv$, S, $\vdash$
ϵ	$\vdash$ b p l q d $\dashv$	$\vdash$ S $\dashv$	Match $\vdash$
$\vdash$	b p l q d $\dashv$	S $\dashv$	pop S, Predict[S][b] gives rule 2 push d, S, b
$\vdash$	b p l q d $\dashv$	b S d $\dashv$	match b
$\vdash$ b	p l q d $\dashv$	S d $\dashv$	pop S, Predict[S][p] gives rule 3 push q, S, p
$\vdash$ b	p l q d $\dashv$	p S q d $\dashv$	match p
$\vdash$ b p	l q d $\dashv$	S q d $\dashv$	pop S, Predict[S][l] gives rule 4 push C
$\vdash$ b p	l q d $\dashv$	C q d $\dashv$	pop C, Predict[C][l] gives rule 5 push C, l
$\vdash$ b p	l q d $\dashv$	l C q d $\dashv$	match l
$\vdash$ b p l	q d $\dashv$	C q d $\dashv$	pop C, Predict[C][q] gives rule 6
$\vdash$ b p l	q d $\dashv$	q d $\dashv$	match q
$\vdash$ b p l q	d $\dashv$	d $\dashv$	match d
$\vdash$ b p l q d	$\dashv$	$\dashv$	match $\dashv$
$\vdash$ b p l q d $\dashv$	ϵ	ϵ	Accept

6.2 Self-practice answer 2

(1) $S' \rightarrow \vdash$ S $\dashv$
(2) S $\rightarrow$ b S d
(3) S $\rightarrow$ p S q
(4) S $\rightarrow$ C
(5) C $\rightarrow$ lC
(6) C $\rightarrow \epsilon$

Predict Table:

	$\vdash$	$\dashv$	b	d	p	q	l
S'	1						
S		4	2	4	3	4	4
C		6		6		6	5

Trace the LL(1) parsing algorithm using the input string $\vdash$ b l b d $\dashv$ using the grammar and predict table just constructed. Your steps should include columns for `Read`, `Unread`, `Stack` and `Action` as shown earlier in this module (Section 2.3).

Read	Unread	Stack	Action
ϵ	$\vdash$ b l b d $\dashv$	S'	pop S', Predict[S'][$\vdash$] gives rule 1 push $\dashv$, S, $\vdash$
ϵ	$\vdash$ b l b d $\dashv$	$\vdash$ S $\dashv$	Match $\vdash$
$\vdash$	b l b d $\dashv$	S $\dashv$	pop S, Predict[S][b] gives rule 2 push d, S, b
$\vdash$	b l b d $\dashv$	b S d $\dashv$	match b
$\vdash$ b	l b d $\dashv$	S d $\dashv$	pop S, Predict[S][l] gives rule 4 push C
$\vdash$ b	l b d $\dashv$	C d $\dashv$	pop C, Predict[C][l] gives rule 5 push C, l
$\vdash$ b	l b d $\dashv$	l C d $\dashv$	match l
$\vdash$ b l	b d $\dashv$	C d $\dashv$	pop C, Predict[C][b] has no rule!

This input generates a parse error; the input string cannot be derived by the grammar!

6.3 Self-practice answer 3

Using the algorithms discussed above, create tables showing the iterations in computing the nullable, First and Follow sets for the following grammar:

(0) S' → ⊢ S ⊣
(1) S → c
(2) S → QRS
(3) Q → R
(4) Q → d
(5) R → ϵ
(6) R → b

Nullable Table:

Iter	0	1	2	3
S'	F	F	F	F
S	F	F	F	F
Q	F	F	T	T
R	F	T	T	T

First Table:

Iter	0	1	2	3
S'	{}	{⊢}	{⊢}	{⊢}
S	{}	{c}	{b, c, d}	{b, c, d}
Q	{}	{d}	{b, d}	{b, d}
R	{}	{b}	{b}	{b}

Follow Table:

Iter	0	1	2
S	{}	{⊣}	{⊣}
Q	{}	{b, c, d}	{b, c, d}
R	{}	{b, c, d}	{b, c, d}

6.4 Self-practice answer 4

For the grammar below (same as the grammar above) compute the Predict Table. The values for nullable, First and Follow are provided:

(0) S'→ ⊢ S ⊣
(1) S → c
(2) S → QRS
(3) Q → R
(4) Q → d
(5) R → ϵ
(6) R → b

	Nullable	First	Follow
S'	False	{⊢}	{}
S	False	{b, c, d}	{⊣}
Q	True	{b, d}	{b, c, d}
R	True	{b}	{b, c, d}

We defined Predict as:
Predict(A, a) = {A → β : a ∈ First(β)} ∪ { A → β: β is nullable and a ∈ Follow(A) }

Predict Table:

	⊢	b	c	d	⊣
S'	{0}				
S		{2}	{1, 2}	{2}	
Q		{3}	{3}	{3, 4}	
R		{5, 6}	{5}	{5}	

Note that this tells us that the grammar is **NOT** LL(1). Consider a derivation for the input ⊢ c ⊣: One derivation is: S ⇒ ⊢ S ⊣ ⇒ ⊢ c ⊣. Another is: S ⇒ ⊢ S ⊣ ⇒ ⊢ QRS ⊣ ⇒ ⊢ RRS ⊣ ⇒ ⊢ RS ⊣ ⇒ ⊢ S ⊣ ⇒ ⊢ c ⊣. The problem is that with S on the stack and our intent to derive c we have choice of using rule 1 or 2 (as shown by the occurrence of both these rules in the predict table).

6.5 Self-practice answer 5

Construct the four tables (Nullable, First, Follow and Predict) for the following grammar

(0) $S' \rightarrow \vdash$ S $\dashv$
(1) S $\rightarrow$ Bb
(2) S $\rightarrow$ Cd
(3) B $\rightarrow$ aB
(4) B $\rightarrow \epsilon$
(5) C $\rightarrow$ cC
(6) C $\rightarrow \epsilon$

	Nullable	First	Follow
S'	False	$\{\vdash\}$	$\{\}$
S	False	$\{a, b, c, d\}$	$\{\dashv\}$
B	True	$\{a\}$	$\{b\}$
C	True	$\{c\}$	$\{d\}$

Predict:

	$\vdash$	a	b	c	d	$\dashv$
S'	0					
S		1	1	2	2	
B		3	4			
C				5	6	

6.6 Self-practice answer 6

Given the following grammar and Predict Table, trace the LL(1) parsing algorithm using the input strings for ⊢ a b ⊣ and ⊢ a b c ⊣:

(0) $S' \rightarrow$ ⊢ S ⊣
(1) S → Bb
(2) S → Cd
(3) B → aB
(4) B → ϵ
(5) C → cC
(6) C → ϵ

Predict:

	⊢	a	b	c	d	⊣
S'	0					
S		1	1	2	2	
B		3	4			
C				5	6	

Read	Unread	Stack	Action
ϵ	⊢ a b ⊣	S'	pop S', Predict[S'][⊢] gives rule 0 push ⊣, S, ⊢
ϵ	⊢ a b ⊣	⊢ S ⊣	Match ⊢
⊢	a b ⊣	S ⊣	pop S, Predict[S][a] gives rule 1 push b, B
⊢	a b ⊣	B b ⊣	pop B, Predict[B][a] gives rule 3 push B, a
⊢	a b ⊣	a B b ⊣	match a
⊢ a	b ⊣	B b ⊣	pop B, Predict[B][b] gives rule 4
⊢ a	b ⊣	b ⊣	match b
⊢ a b	⊣	⊣	match ⊣
⊢ a b ⊣	ϵ	ϵ	Accept

Read	Unread	Stack	Action
ϵ	⊢ a b c ⊣	S'	pop S', Predict[S'][⊢] gives rule 1 push ⊣, S, ⊢
ϵ	⊢ a b c ⊣	⊢ S ⊣	Match ⊢
⊢	a b c ⊣	S ⊣	pop S, Predict[S][a] gives rule 1 push b, B
⊢	a b c ⊣	B b ⊣	pop B, Predict[B][a] gives rule 3 push B, a
⊢	a b c ⊣	a B b ⊣	match a
⊢ a	b c ⊣	B b ⊣	pop B, Predict[B][b] gives rule 4
⊢ a	b c ⊣	b ⊣	match b
⊢ a b	c ⊣	⊣	TOS does not match next input symbol

Notice how this parse fails because the TOS is ⊣ whereas we are trying to match the symbol c.