Follow Sets

Michael Goodrich and Brett Decker

September 2020

1 Introduction

Professor Decker and I are following a pattern of learning activities for each class. The pattern is made up of a sequence of four steps:

1. Have you read formal definitions of the relevant mathematical concepts.
2. Have you view a video that applies the definitions in either an example problem or to a project.
3. Have you participate in a "short" zoom lecture where you can get clarification about things from the definition or example that are tricky.
4. Have you work in small groups on either homework or projects where you apply the ideas.

This tutorial gives a formal definition of FOLLOW sets, discusses what the notation in the definition means, gives an example, and discusses how FOLLOW sets are used in a parse table.

2 Definition and Discussion

2.1 Running Example

Let's begin with a running example. Consider a grammar with terminals

$$T = \{+, *, (,), 1, 2, 3\},$$

nonterminals

$$N = \{E, A, B, C, D\},$$

starting nonterminal E, and productions

$$\begin{array}{rcl} E & \rightarrow & AC \\ C & \rightarrow & +E|\lambda \\ A & \rightarrow & BD \\ D & \rightarrow & *A|\lambda \\ B & \rightarrow & (E)|1|2|3 \end{array}$$

The grammar allows us to derive simple mathematical expressions.

The FIRST sets for some of the productions are given below:

Production	**FIRST**
$E \to AC$	$\{(, 1, 2, 3\}$
$C \to +E$	$\{+\}$
$C \to \lambda$	$\{\lambda\}$
$A \to BD$	$\{(, 1, 2, 3\}$
$D \to *A$	$\{*\}$
$D \to \lambda$	$\{\lambda\}$.

2.2 Review of Definitions from Textbook

Recall from Definition 2 in Section 13.1.2 of the textbook that a *phase-structured grammar* has four elements: a vocabulary, a subset of the vocabulary that we call *terminals*, a start symbol, and a finite set of productions. A set of *nonterminal* symbols is also defined as everything in the vocabulary that is not a terminal. The start symbol is an element of the nonterminal subset. The vocabulary set, terminal set, nonterminal set, starting symbol, and production set are represented by V, T, N, S, and P, respectively.

Types of grammars were defined in the first three paragraphs of Section 13.1.3 from the textbook. We don't talk about **type 0** or **type 1** grammars in this class. We didn't talk much about **type 3**, which are called *regular grammars* because we chose to use *regular expressions* to represent regular grammars rather that using the formal definition. Consequently, our discussion of grammars focuses on **type 2** grammars, which are called *context-free grammars*.

We decided that we wanted to be able to parse without every having to backtrack, so we restricted attention to *LL(1) grammars*, which are grammars where we can always tell which production to apply given a nonterminal at the top of the stack and the current input character in the input stream. This tutorial is about an important definition of LL(1) grammars, namely the *FOLLOW* set.

Recall further that the single arrow, $\to$ is used in a production to tell us how to change a nonterminal into something else. That something else is made up of a concatenation of terminals and nonterminals. For example, from the grammar from Section 2.1 we see $\to$ in each production.

The double arrow, $\Rightarrow$ denotes one step in a derivation. It is used in listing sequences of derivation steps, such as in the leftmost derivation of the string $1 + 2$.

Example 1 *Using double arrow in the derivation of* $1 + 2$.

$$\begin{aligned} E &\Rightarrow AC && (1)\\ &\Rightarrow BDC && (2)\\ &\Rightarrow 1DC && (3)\\ &\Rightarrow 1C && (4) \end{aligned}$$

$$\Rightarrow \quad 1 + E \tag{5}$$
$$\Rightarrow \quad 1 + AC \tag{6}$$
$$\Rightarrow \quad 1 + BDC \tag{7}$$
$$\Rightarrow \quad 1 + 2DC \tag{8}$$
$$\Rightarrow \quad 1 + 2C \tag{9}$$
$$\Rightarrow \quad 1 + 2. \tag{10}$$

Step(3) turns into step (4) via the production $D \to \lambda$. A production involving a λ is also used between step (8) and step (9), and between step (9) and step (10).

The double arrow with a star $\stackrel{*}{\Rightarrow}$ denotes a derivation with zero or more steps. For example, $E \stackrel{*}{\Rightarrow} 1 + 2$ represents all the steps in the derivation just given. The partial derivation from step (1) to step (7) could be written $E \stackrel{*}{\Rightarrow} 1 + BDC$.

2.3 FOLLOW Set Definition

A FOLLOW set can be defined for every nonterminal, but it is only necessary to compute the FOLLOW set for some of the nonterminals. We'll talk about those nonterminals later.

Definition 1 *For the grammar (V, T, S, P), the* FOLLOW set *of a nonterminal $A \in N$ is given by*

$$FOLLOW(A) = \begin{cases} \{t \in T : S \stackrel{*}{\Rightarrow} \omega A t \eta\} \bigcup \{\#\} & \text{if } S \stackrel{*}{\Rightarrow} \alpha A \\ \{t \in T : S \stackrel{*}{\Rightarrow} \omega A t \eta\} & \text{otherwise} \end{cases} \tag{11}$$

The symbol # denotes the "end of string character".

When I first saw Equation (11), I had no idea what it meant. I figure that many of you feel the same way, so let's break it down into pieces.

- FOLLOW(A) can be one of two sets depending on the condition shown in the right side of the equation. Both conditions have one set in common, so let's talk about that first.
- If the definition said $t \in T$, the definition would mean that t represents a single **terminal**.
- The curly braces indicate that the FOLLOW set is a set of zero or more terminals.
- The colon is read as "such that", so the set in Equation (11) means that the only terminals that make it into the FOLLOW set of A must satisfy the condition on the right side of the colon.
- The mess to the right of the colon, $S \stackrel{*}{\Rightarrow} \omega A t \eta$, says that it is possible to derive $\omega A t \eta$ in zero or more steps from the starting nonterminal S.

- The messy stuff in the string $\omega At\eta$ has four parts:
  1. ω at the left is a lower case Greek letter "omega". It represents a string that can be made up of both terminals or nonterminals. It is possible for $\omega = \lambda$.
  2. η at the right is also a lower case Greek letter "eta". It also represents a string that can be made up of both terminals or nonterminals. It is possible for $\eta = \lambda$.
  3. The A is the nonterminal for which we are computing the FOLLOW set. Notice that the definition computes FOLLOW(A).
  4. The t is a terminal that can "follow" the A in some derivation from the starting nonterminal.
- The $\cup$ indicates the *union* operator. The top line says to include the "end of string" character in the first set under the special condition that it is possible to derive something with an A in the rightmost spot.

Stated succinctly, the FOLLOW set of a nonterminal is the set of terminals that can appear to the *right* of that nonterminal in some derivation from the starting nonterminal.

We're pretty sure that this doesn't make a lot of sense yet, so let's give examples of $S \stackrel{*}{\Rightarrow} \omega At\eta$ using the grammar from Section 2.1. We'll do this in the reverse order that you'd usually do it, giving you the FOLLOW set and then showing how the elements of the FOLLOW set satisfy the definition.

2.4 Examples

Let's repeat the productions here for convenience.

$$\begin{aligned} E &\rightarrow AC \\ C &\rightarrow +E|\lambda \\ A &\rightarrow BD \\ D &\rightarrow *A|\lambda \\ B &\rightarrow (E)|1|2|3 \end{aligned}$$

The starting nonterminal is E.

Example 2 $FOLLOW(C) = \{), \#\}$

Let's show a derivation

$$\begin{aligned} E &\Rightarrow AC \\ &\Rightarrow BDC \\ &\Rightarrow (E)DC \\ &\Rightarrow (AC)DC \end{aligned}$$

Thus, $E \stackrel{*}{\Rightarrow} (AC)DC$.

Let's break the right hand side of the last line into chunks that match the pattern from the definition, $S \stackrel{*}{\Rightarrow} \omega C t \eta$. Let

$$\begin{aligned} \omega &= (A \\ t &=) \\ \eta &= DC. \end{aligned}$$

The $(AC)DC = \omega C t DC$, as desired.

In summary, we showed a derivation in which the closing parenthesis ')' "followed" C, so ')'$\in$ FOLLOW(C).

We can now show that $\# \in$ FOLLOW(C).

$$E \Rightarrow AC$$

This is where that strange $\cup\{\#\}$ applies. We showed that $E \Rightarrow AC$, and therefore that $E \stackrel{*}{\Rightarrow} AC$. This matches the top condition in Equation (11) with $\alpha = A$.

The meaning of $\{\#\} \in$ FOLLOW(C) is pretty simple, even though the notation is strange. It simply means that it is possible to create a derivation where it is legal for the derivation to terminate with a C. This makes sense because we can see that $C \rightarrow \lambda$, so it's possible to derive a string where the last thing we want C to produce is λ, which is what occurs when the input string has nothing more to find.

Example 3 *FOLLOW*(D) $= \{+,), \#\}$.

We can show this using three partial derivations. Let's begin with '+'$\in$ FOLLOW(D).

$$\begin{aligned} E &\Rightarrow AC \\ &\Rightarrow A+E \\ &\Rightarrow BD+E \end{aligned}$$

Thus, $E \stackrel{*}{\Rightarrow} BD+E$, which matches the pattern with $\omega = B$, $t =$'+', and $\eta = E$.

Let's now show that ')'$\in$ FOLLOW(D).

$$\begin{aligned} E &\Rightarrow AC \\ &\Rightarrow A \\ &\Rightarrow BD \\ &\Rightarrow B \\ &\Rightarrow (E) \\ &\Rightarrow (AC) \\ &\Rightarrow (BDC) \\ &\Rightarrow (BD). \end{aligned}$$

Thus, $E \stackrel{*}{\Rightarrow} (BD)$, which matches the pattern with $\omega = (B,\ t =')'$, and $\eta = \lambda$.

Finally, let's show that '#'$\in$ FOLLOW(D).

$$\begin{aligned} E &\Rightarrow AC \\ &\Rightarrow A \\ &\Rightarrow BD \end{aligned}$$

Thus, $E \stackrel{*}{\Rightarrow} BD$, which matches the pattern on the first line of Equation (11), with $\alpha = B$.

2.5 An Algorithm for Computing FOLLOW Sets

There exists an algorithm for computing FOLLOW sets. That's good because the examples that I just showed seemed to be pulled out of thin air. The algorithm isn't hard to understand, but it's not essential for Project 2 since the grammar used in Project 2 allows FOLLOW sets to be computed really easily. Consequently, we won't discuss the algorithm in this class. See the related video on the course schedule, which shows how to compute the FOLLOW sets for Project 2.

2.6 Application to Parse Tables

We used FIRST sets to determine which productions belonged in which cells in the parse table. Similarly, we use FOLLOW sets to determine when we should *pop the stack without advancing the input.* Let's illustrate for two of the productions in the grammar,

$$\begin{aligned} C &\rightarrow +E|\lambda \\ D &\rightarrow *A|\lambda \end{aligned}$$

Let's sketch the parse table but only fill out a rows for the nonterminals C and D as well as for a few of the terminals.

	Input String				
Stack	+	*	(	)	#
C					
D					
+	AdPop				
*		AdPop			
(			AdPop		
)				AdPop	
#					accept

The FIRST sets are FIRST($C \rightarrow +E$) = {+} and FIRST($D \rightarrow *A$) = {*}. Adding this information to the table yields

Stack	Input String				
	+	*	(	)	#
C	+E				
D		*A			
+	AdPop				
*		AdPop			
(			AdPop		
)				AdPop	
#					accept

The FOLLOW sets are FOLLOW$(C) = \{), \#\}$ and FOLLOW$(D) = \{+,), \#\}$. The key idea is that we use the FOLLOW sets to determine what to do with productions of the form $C \to \lambda$ or $D \to \lambda$.

Consider the two productions for nonterminal C

$$\begin{aligned} C &\to +E \\ C &\to \lambda \end{aligned}$$

If the top of the stack is C and the current input character is an element of the FIRST set, then I know that I should apply the production $C \to +E$, which means that I pop the C from the stack and replace it with $+E$. The FOLLOW set tells me when I should use the production $C \to \lambda$. The idea is that if C is at the top of the stack and I see an element of the FOLLOW set, then I should stop using C and move to the next part of my parse. This is equivalent to simply popping the stack. Thus, for each element of the set FOLLOW(C), I simply pop the stack. I don't advance the input. The cell in the parse table is labeled **Pop**, indicating that the stack is popped but the input is not advanced.

The parse table becomes

Stack	Input String				
	+	*	(	)	#
C	+E			**Pop**	**Pop**
D	**Pop**	*A		**Pop**	**Pop**
+	AdPop				
*		AdPop			
(			AdPop		
)				AdPop	
#					accept