PROJECT 2: Useless Symbols, FIRST and FOLLOW sets, and Predictive Parsing

CSE 340 FALL 2020

Rida A. Bazzi

Project 2 Goals

- I have introduced in class predictive parsing and FIRST and FOLLOW sets
- The goal of this project is to show you how the process of building a predictive parser can be automated
- Another important goal of the project is to give you experience in writing a substantial program which is non-trivial conceptually
  - This will make you a better programmer
  - You will have a better understanding of the power of abstraction in building code
  - You will have a better appreciation of the material covered so far

Outline

- Set Operations
- Grammar representation
- Calculating useless symbols
- Calculating FIRST sets
- Calculating FOLLOW sets
- Determining if a grammar has a predictive parser

Set Operations

- In calculating FIRST and FOLLOW sets, you need to represent theses sets as a data structure in your program and you need to do operations on these sets
- The operations you need are
  - $A = A \cup (B - \{\varepsilon\})$: Adding the elements of one set B with the exception of epsilon to another set A and check if the set changed due to the additions
  - $A = A \cup \{\varepsilon\}$: Adding epsilon to a set and check if the set changed due to the addition
  - is_epsilon_in(A): Checking if epsilon belongs to a set
  - printing the elements of a set according to some order

I suggest that you write a function for each of these functionalities (and others you might identify) to make your code easier to work with

Set Operations and keeping track of change

- C++ has a number of libraries and data structures that can allow you to define sets. You should look at those and adopt one of them
- I comment on keeping track of change when adding elements of set S1 to set S2. Here is the pseudocode

```
for every element in S1 that is not epsilon
        if element is not in S2
                changed = true
                add element to S2
```

In the pseudocode, changed is a Boolean variable. I described how it is used in the typed notes on FIRST and FOLLOW.

- I think that you should have all the functions for set operation in place before you attempt to write higher-level functionality. You will end up fighting less with your code

Referring to Sets

- Another functionality you need in calculating FIRST and FOLLOW sets is the ability to refer to something like FIRST(A) or FOLLOW(B). The rules numbered with roman numerals I through V for FIRST and FOLLOW sets that we have seen in class assume you can do that
- In your program, you will need to
  - represent terminals and non-terminals
  - refer to the sets (FIRST and FOLLOW) of particular terminals and non-terminals
- A common approach I saw students use is to represent terminals and non-terminals as strings (remember your program will read the names of terminals and non-terminals as IDs and the lexeme string is the name)

Representing Terminals and Non-Terminals

- You should read all terminals and non-terminals as strings and store them in a list that I will call universe or (symbols). The universe will include representations for epsilon ("#") and EOF ("$")

- In order to be able to refer to FIRST(A), you can use the index of A in the list, so you can say FIRST[Index(A)], where index (A) is a function that takes a string as a parameter and returns its index in the list

- Alternatively, you can use an unordered map for FIRST sets and another one for FOLLOW sets and refer to FIRST[A] and FOLLOW[A], where A is a string. You should lookup how to use unordered maps if you want to follow this approach

- Alternatively, you can have a more efficient implementation in terms of space and performance. You can store the indices and not the strings when representing grammar rules. This will effectively replace every symbol with an integer index which allows you to use FIRST[A_{index}] where A is the index for A

- Let us see how this can be done and then we get back to FIRST and FOLLOW

Grammar Representation

S → A B C (1)
A → D E (2)
B → b B (3)
B → ε (4)
C → c C (5)
C → ε (6)
D → d D (7)
D → ε (8)
E → e E (9)
E → ε (10)

symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

Rule 1

LHS 6
RHS 7 8 9
RHS_size: 3

Grammar Representation

S → A B C	(1)
A → D E	(2)
B → b B	(3)
B → ε	(4)
C → c C	(5)
C → ε	(6)
D → d D	(7)
D → ε	(8)
E → e E	(9)
E → ε	(10)

symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

Rule 2

LHS 7

RHS 10 11

RHS_size: 2

Grammar Representation

S	→ A B C	(1)
A	→ D E	(2)
B	→ b B	(3)
B	→ ε	(4)
C	→ c C	(5)
C	→ ε	(6)
D	→ d D	(7)
D	→ ε	(8)
E	→ e E	(9)
E	→ ε	(10)

symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

Rule 2

LHS 8

RHS 2 8

RHS_size: 2

Grammar Representation

S → A B C (1)
A → D E (2)
B → b B (3)
B → ε (4)
C → c C (5)
C → ε (6)
D → d D (7)
D → ε (8)
E → e E (9)
E → ε (10)

symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

Rule 2

LHS 8
RHS 0
RHS_size: 1

Grammar Representation Example

$S \rightarrow A\ B\ C$ (1)
$A \rightarrow D\ E$ (2)
$B \rightarrow b\ B$ (3)
$B \rightarrow \varepsilon$ (4)
$C \rightarrow c\ C$ (5)
$C \rightarrow \varepsilon$ (6)
$D \rightarrow d\ D$ (7)
$D \rightarrow \varepsilon$ (8)
$E \rightarrow e\ E$ (9)
$E \rightarrow \varepsilon$ (10)

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

LHS: 6	RHS: 7, 8, 9
LHS: 7	RHS: 10, 11
LHS: 8	RHS: 2 , 8
LHS: 8	RHS: 0
LHS: 9	RHS: 3, 9
LHS: 9	RHS: 0
LHS: 10	RHS: 4, 10
LHS: 10	RHS: 0
LHS: 11	RHS: 5, 11
LHS: 11	RHS: 0

Rules

Grammar Representation

- You need a list of all the rules: this can simply be a vector of rules
- Every rules has a LHS which is an integer index
- Every rule has a RHS which is a vector of integers, one integer for every symbol on the RHS
- To put the LHS and RHS together you can declare a structure with two fields, one for the LHS and one for the RHS

Iterating over grammar representation

- Once you have a vector of rules, you can easily iterate over all the rules
- Also, for a given rule, you can easily iterate over the RHS
- For calculating FIRST sets (see later also), you can now refer to FIRST[rule.LHS] or FIRST[rule.RHS[j]], which is more convenient than writing FIRST[index(rule.LHS)] and FIRST[index(rule.RHS[j])]
- Having all entries as integer indices makes the code easier to work with
- The strings (names of various symbols) are only needed when the output is produced. To print a symbol whose index is A, you simply print Symbols[A].

Useless Symbols

A symbol is useless if it does not appear in the derivation of a string of terminals or in the derivation of the empty string

A symbol is not useless if it appears in the derivation of a string of terminals or in a derivation of the empty string

$$S \overset{*}{\Rightarrow} xAy \overset{*}{\Rightarrow} w \in T^*$$

Calculating Useless Symbols

1. We start by calculating generating symbols
   - A symbol A is generating if it can derive a string in T^* (sequence of zero or more terminals)

$$A \overset{*}{\Rightarrow} w \in T^*$$

   - At the end of this step, you should remove any grammar rule that has a non-generating symbol

2. Then we determine reachable symbols
   - A symbol A is reachable if S can derive a sentential form containing A:

$$S \overset{*}{\Rightarrow} x\,A\,y$$

   - At the end of this step, you should remove all grammar rules that have non-reachable symbols

The order given is important. The calculation should be done in the order given: Calculating reachable first, then calculating generating does not work

Calculating generating symbols

1. Initialization
   - all terminals are generating
   - ε is generating

2. If $A \rightarrow A_1 A_2 \ldots A_k$ is a grammar rule and
   - A_1 generating and
   - A_2 generating and
   - … and
   - …
   - A_k generating

A is generating

Iterative approach to calculating generating symbols

$S \rightarrow A\ B\ C$ (1)
$A \rightarrow D\ E$ (2)
$B \rightarrow b\ B$ (3)
$B \rightarrow \varepsilon$ (4)
$C \rightarrow c\ C$ (5)
$C \rightarrow \varepsilon$ (6)
$D \rightarrow d\ D$ (7)
$D \rightarrow \varepsilon$ (8)
$E \rightarrow e\ E$ (9)
$E \rightarrow \varepsilon$ (10)

Symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

Generating array

0	F
1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F
9	F
10	F
11	F

Iterative approach to calculating generating symbols: Initialization

S → A B C (1)
A → D E (2)
B → b B (3)
B → ε (4)
C → c C (5)
C → ε (6)
D → d D (7)
D → ε (8)
E → e E (9)
E → ε (10)

Symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

Generating array

0	T
1	F
2	T
3	T
4	T
5	T
6	F
7	F
8	F
9	F
10	F
11	F

Iterative approach to calculating generating symbols: Iteration

S → A B C	(1)
A → D E	(2)
B → b B	(3)
B → ε	(4)
C → c C	(5)
C → ε	(6)
D → d D	(7)
D → ε	(8)
E → e E	(9)
E → ε	(10)

Symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

- Since A, B, and C are not known to be generating, we cannot say that S is generating, so there is no change

Generating array

0	T
1	F
2	T
3	T
4	T
5	T
6	F
7	F
8	F
9	F
10	F
11	F

Iterative approach to calculating generating symbols: Iteration

$S \to A\ B\ C$ (1)
$A \to D\ E$ (2)
$B \to b\ B$ (3)
$B \to \varepsilon$ (4)
$C \to c\ C$ (5)
$C \to \varepsilon$ (6)
$D \to d\ D$ (7)
$D \to \varepsilon$ (8)
$E \to e\ E$ (9)
$E \to \varepsilon$ (10)

Symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

- Since D and E are not know to be generating, we cannot say that A is generating, so there is no change

Generating array

0	T
1	F
2	T
3	T
4	T
5	T
6	F
7	F
8	F
9	F
10	F
11	F

Iterative approach to calculating generating symbols: Iteration

S → A B C	(1)
A → D E	(2)
B → b B	(3)
B → ε	(4)
C → c C	(5)
C → ε	(6)
D → d D	(7)
D → ε	(8)
E → e E	(9)
E → ε	(10)

Symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

- Here b is know to be generating (we can determine that by checking the array), but B is not known to be generating, so , again there is no change.
- Note that we do not care that the symbol B appears on the left and right sides. We just do the same check for all the rules.

Generating array

0	T
1	F
2	T
3	T
4	T
5	T
6	F
7	F
8	F
9	F
10	F
11	F

Iterative approach to calculating generating symbols: Iteration

S → A B C (1)
A → D E (2)
B → b B (3)
B → ε (4)
C → c C (5)
C → ε (6)
D → d D (7)
D → ε (8)
E → e E (9)
E → ε (10)

Symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

- Here, since ε is generating, we conclude that B is generating

Generating array

0	T
1	F
2	T
3	T
4	T
5	T
6	F
7	F
8	F
9	F
10	F
11	F

Iterative approach to calculating generating symbols: Iteration

S → A B C (1)
A → D E (2)
B → b B (3)
B → ε (4)
C → c C (5)
C → ε (6)
D → d D (7)
D → ε (8)
E → e E (9)
E → ε (10)

Symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

- B's entry changes to true

Generating array

0	T
1	F
2	T
3	T
4	T
5	T
6	F
7	F
8	T
9	F
10	F
11	F

Iterative approach to calculating generating symbols: Iteration

$S \rightarrow A\ B\ C$ (1)
$A \rightarrow D\ E$ (2)
$B \rightarrow b\ B$ (3)
$B \rightarrow \varepsilon$ (4)
$C \rightarrow c\ C$ (5)
$C \rightarrow \varepsilon$ (6)
$D \rightarrow d\ D$ (7)
$D \rightarrow \varepsilon$ (8)
$E \rightarrow e\ E$ (9)
$E \rightarrow \varepsilon$ (10)

Symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

- No change

Generating array

0	T
1	F
2	T
3	T
4	T
5	T
6	F
7	F
8	T
9	F
10	F
11	F

Iterative approach to calculating generating symbols: Iteration

S → A B C (1)
A → D E (2)
B → b B (3)
B → ε (4)
C → c C (5)
C → ε (6)
D → d D (7)
D → ε (8)
E → e E (9)
E → ε (10)

Symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

- C is generating

Generating array

0	T
1	F
2	T
3	T
4	T
5	T
6	F
7	F
8	T
9	F
10	F
11	F

Iterative approach to calculating generating symbols: Iteration

$S \rightarrow A\,B\,C$ (1)
$A \rightarrow D\,E$ (2)
$B \rightarrow b\,B$ (3)
$B \rightarrow \varepsilon$ (4)
$C \rightarrow c\,C$ (5)
$C \rightarrow \varepsilon$ (6)
$D \rightarrow d\,D$ (7)
$D \rightarrow \varepsilon$ (8)
$E \rightarrow e\,E$ (9)
$E \rightarrow \varepsilon$ (10)

Symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

- C's entry changes to true

Generating array

0	T
1	F
2	T
3	T
4	T
5	T
6	F
7	F
8	T
9	T
10	F
11	F

Iterative approach to calculating generating symbols: Iteration

S → A B C	(1)
A → D E	(2)
B → b B	(3)
B → ε	(4)
C → c C	(5)
C → ε	(6)
D → d D	(7)
D → ε	(8)
E → e E	(9)
E → ε	(10)

Symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

- At the end of the first round (going over all rules), we get the array on the right
- Since some entries have changed, we need to do another round

Generating array

0	T
1	F
2	T
3	T
4	T
5	T
6	F
7	F
8	T
9	T
10	T
11	T

Iterative approach to calculating generating symbols: Iteration

S → A B C	(1)
A → D E	(2)
B → b B	(3)
B → ε	(4)
C → c C	(5)
C → ε	(6)
D → d D	(7)
D → ε	(8)
E → e E	(9)
E → ε	(10)

Symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

- We examine the first rule again, but we cannot tell that S is generating because, even though B and C are generating, A is not known to be generating

Generating array

0	T
1	F
2	T
3	T
4	T
5	T
6	F
7	F
8	T
9	T
10	T
11	T

Iterative approach to calculating generating symbols: Iteration

S → A B C	(1)	
A → D E	(2)	
B → b B	(3)	
B → ε	(4)	
C → c C	(5)	
C → ε	(6)	
D → d D	(7)	
D → ε	(8)	
E → e E	(9)	
E → ε	(10)	

Symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

- We examine the second rule and now we can tell that A is generating because every symbol on the RHS of the rule for A is generating.

Generating array

0	T
1	F
2	T
3	T
4	T
5	T
6	F
7	F
8	T
9	T
10	T
11	T

Iterative approach to calculating generating symbols: Iteration

$S \rightarrow A\ B\ C$ (1)
$A \rightarrow D\ E$ (2)
$B \rightarrow b\ B$ (3)
$B \rightarrow \varepsilon$ (4)
$C \rightarrow c\ C$ (5)
$C \rightarrow \varepsilon$ (6)
$D \rightarrow d\ D$ (7)
$D \rightarrow \varepsilon$ (8)
$E \rightarrow e\ E$ (9)
$E \rightarrow \varepsilon$ (10)

Symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

- So we change A's entry to true

Generating array

0	T
1	F
2	T
3	T
4	T
5	T
6	F
7	T
8	T
9	T
10	T
11	T

Iterative approach to calculating generating symbols: Iteration

$S \rightarrow A\ B\ C$ (1)
$A \rightarrow D\ E$ (2)
$B \rightarrow b\ B$ (3)
$B \rightarrow \varepsilon$ (4)
$C \rightarrow c\ C$ (5)
$C \rightarrow \varepsilon$ (6)
$D \rightarrow d\ D$ (7)
$D \rightarrow \varepsilon$ (8)
$E \rightarrow e\ E$ (9)
$E \rightarrow \varepsilon$ (10)

Symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

- The remaining rules do not result in any change
- But since some entries have changed in the second round, we need to do a third round

Generating array

0	T
1	F
2	T
3	T
4	T
5	T
6	F
7	T
8	T
9	T
10	T
11	T

Iterative approach to calculating generating symbols: Iteration

S → A B C (1)
A → D E (2)
B → b B (3)
B → ε (4)
C → c C (5)
C → ε (6)
D → d D (7)
D → ε (8)
E → e E (9)
E → ε (10)

Symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

- In the third round, we determine that S is generating because all the symbols on the RHS of the rule S →ABC are generating and the entry for S is changed to true.
- Since some entries changed in the third round, we need to do a fourth round

Generating array

0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T

Iterative approach to calculating generating symbols: Iteration

$S \rightarrow A\ B\ C$ (1)
$A \rightarrow D\ E$ (2)
$B \rightarrow b\ B$ (3)
$B \rightarrow \varepsilon$ (4)
$C \rightarrow c\ C$ (5)
$C \rightarrow \varepsilon$ (6)
$D \rightarrow d\ D$ (7)
$D \rightarrow \varepsilon$ (8)
$E \rightarrow e\ E$ (9)
$E \rightarrow \varepsilon$ (10)

Symbols

0	1	2	3	4	5	6	7	8	9	10	11
"#"	"$"	"b"	"c"	"d"	"e"	"S"	"A"	"B"	"C"	"D"	"E"

- In the fourth round nothing changes and we have our answer

Generating array

0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T

Removing rules with non-generating symbols

- After we calculate generating symbols, we remove all rules that have a symbol that is not generating
- One way to do this is the following. We iterate over all the rules in the vector of rules
  - For each rule,
    - if every symbol in the rule is generating, push the rule to a new vector.
    - If some symbol in the rule is not generating go do the next rule

At the end, the new vector, let us call it RulesGen contains all the grammar rules with generating symbols.

Calculating Useless Symbols

- We start by calculating generating symbols
  - A symbol is generating if it can derive a string in T^* (zero or more sequence of terminals)
- Then we remove all rules that have a symbol that is not generating
- We have now a new set of rules, which is RulesGen that I mentioned on the previous slide
- Then we start with the RulesGen vector to determine reachable symbols
  - A symbol A is reachable if S can derive a sentential form containing the symbol:

$$S \overset{*}{\Rightarrow} x A y$$

Calculating reachable symbols

1. S is reachable

2. If $A \rightarrow A_1\, A_2 \ldots A_k$ is a grammar rule and A is reachable

$\Rightarrow$ A_1 and A_2 and … and A_k are reachable

Calculating reachable symbols

- Calculation can be done in a way that is similar to how we did generating symbols
- At the end, we have a Boolean array indicating which symbols are reachable
- We remove all rules that have a non-reachable symbol

Things to think about

- You should decide on the data structures you will be using. Things you need to represent are
  - initial list of non-terminals
  - initial list of terminals
  - you should think about how these lists will be used for the various tasks and if they need to be combined into a larger list of symbols
  - grammar rules: LHS, RHS
  - Set representation. You should think about the operation you will need to be doing on sets
- Before you start coding, you should have an outline of how you will be using your data structures to implement the various tasks
- Before you start coding, make you you have a correct understanding of the requirements
- I and the TAs will be happy to look at your initial outline of how you will approach the project to give you feedback
- When you start coding, we will be happy to look at your code to give you feedback. The earlier you ask the better off you will be.