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

CSE 340 FALL 2021

Rida A. Bazzi

Project 3 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

- The big picture
- The foundation: Set Operations
- Reading the Grammar
- Task 1: Reading the grammar
- Grammar Representation
- Task 2: Calculating useless symbols

- This presentation does not replace the project description. The project description is the ultimate reference as far as the requirements of the project are concerned
- This presentation is meant to give you an overview and give some specifics about how to implement some functionality
- For the exact requirements, consult the project specification document which is self-contained as far as the requirements are concerned

The Big Picture: Input (1/6)

- In this project, the input is a grammar description.
  - For example, the following is a description of a grammar with three rules, three non-terminal symbols and two terminal symbols (each rule is terminated with *):

```
A -> C B * B -> b * C -> c * #
```

- Your program will read the grammar description and will store it internally in a data structure
  - For example, the grammar above can be represented as

LHS	RHS	
“A”	“C”	“B”
“B”	“b”	
″C”	“c”	

The Big Picture: Task 1 (2/6)

- Task 1. After reading the grammar, your program should determine the terminals and the non-terminals and print them in the order in which they appear in the grammar, non-terminals first then terminals

  For example, for the input: `A -> C B * B -> b * C -> c * #`

  Your program will print

  ```
  b c A C B
  ```

  Notice how C is printed before B before it appears before B in the grammar (first rule for A)

The Big picture: Task 2 Useless symbols (3/6)

For this task, your program should determine the useless symbols, if any, and print the resulting grammar after removing the useless symbols.

We have not defined useless symbols in class yet. The details of the calculations are give later in this presentation.

Informally, a symbol is useless if it does not appear in the derivation of a sequence of terminals.

For example, consider the grammar :

```
A -> C B * A -> D C E * B -> b * C -> c * D -> d E * E -> e D * #
```

In this grammar D and E cannot appear in the derivation of a sequence of terminals, so the output will be

```
A ->  C  B
B -> b
C -> c
```

The Big picture: Task 3 FIRST sets (4/6)

For this task, your program will read the grammar description and prints the FIRST sets for every non-terminal of the grammar.

For example, for the following grammar

```
A -> C B * A -> * A -> D C E * B -> b * C -> c * D -> d E * E -> e D * #
```

Your program will print

```
FIRST(A) =  { # , c }
FIRST(B) =  { b }
FIRST(C) =  { c }
FIRST(D) =  { d }
FIRST(E) =  { e }
```

Notice how `FIRST(D)` and `FIRST(E)` are calculated normally even though they are useless symbols. The calculation is done normally following the algorithm that we covered in class.

Also, `FIRST(A)` contains epsilon which is represented as `#` in the output. The particular order of the contents of the FIRST sets matters. The order is described in the project specification document.

The Big picture: Task 4 FOLLOW sets (5/6)

For this task, your program will read the grammar description and prints the FOLLOW sets for every non-terminal of the grammar.

For example, for the following grammar

```
A -> C B * A -> * A -> D C E * B -> b * C -> c * D -> d E * E -> e D * #
```

Your program will print

```
FOLLOW(A) =  { $ }
FOLLOW(B) =  { $ }
FOLLOW(C) =  { b , e }
FOLLOW(D) =  { c }
FOLLOW(E) =  { $ }
```

The Big picture: Task 5 Determining if Grammar has a Predictive Parser (6/6)

For this task, your program will determine if the grammar has a predictive parser

if the grammar has useless symbols, the output is NO

if the grammar has no useless symbols, the output is

NO if the grammar has no predictive parse
YES if the grammar has a predictive parser

to make the determination, the program should first determine if the grammar has useless symbols. If there are useless symbols, it outputs NO. If there are no useless symbols, then it should

Advice

- Now that I have described the various tasks at a high level, in the remainder of this document I am going to address some implementation and algorithmic issues
- You are not required to follow exactly what I am describing. Only the output matters for grading.
- I strongly recommend that
  - you think carefully about the project and how the pieces will fit together
  - come up with basic functionality that you can build on
  - think in higher-level terms before getting into the details

The Foundation: 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 ∪ (B – {Ɛ }) : 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 ∪ {Ɛ }: 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 can 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  x in S1 that is not epsilon
        if x is not in S2
                changed = true
                add x 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

Reading the Grammar (1/2)

One thing that can be confusing about this project is that

- there is a grammar that describes the input format (see project specification document)
- the input itself represents a grammar!

- The grammar that represents the input format is really simple. Nevertheless, I strongly suggest that you write a proper parser for it. This will result in cleaner code

- When you parse the grammar, you are going to read one rule at a time. Each rule has two parts the `LHS` and the `RHS`. The `LHS` is an `ID` and the `RHS` is a vector of `ID`s, so a rule can be represented as a struct with two fields
  - `LHS: string`
  - `RHS: vector<string>`

Reading the grammar (2/2)

- The first step is to simply read all the rules as described in the previous page and storing them in a vector of rules. This will look like this

LHS	RHS	
“A”	“C”	“B”
“B”	“b”	
”C”	“c”	

for the grammar `A -> C B * B -> b * C -> c * #`

At this point the grammar is read and you can start on Task 1

Task 1: Terminals and Non-Terminals

- This task is not really hard but it has been my experience that sometimes it is made harder than it should be.

- The way, I would approach it is to do two passes over the grammar (I know that it can be done in one pass)
  - first pass: determine the non-terminals by going over the LHSs of the rules. The list you get will not be in the order in which they appear in the grammar (think about the example on slide 6)
  - second pass: determine the terminals and non-terminals in the order in which they appear in the grammar. For this you will go over each rule LHS first then RHS

- At the end of this you will have to lists of terminals and non-terminals in the order in which they appear in the grammar and you can print them

Grammar Representation for the Remaining Tasks

- Before continuing with the other tasks, you need to decide on a representation of the grammar that you can use with the various algorithms
- 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) and to keep the grammar represented as shown on slide 17
- This approach can work, but you need to know what else you need to make it work
- In particular, one 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 that you can do that
- Summary: 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

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. This is not efficient, but it work!

- 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. This is an improvement!

- 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_{index} is the index for A

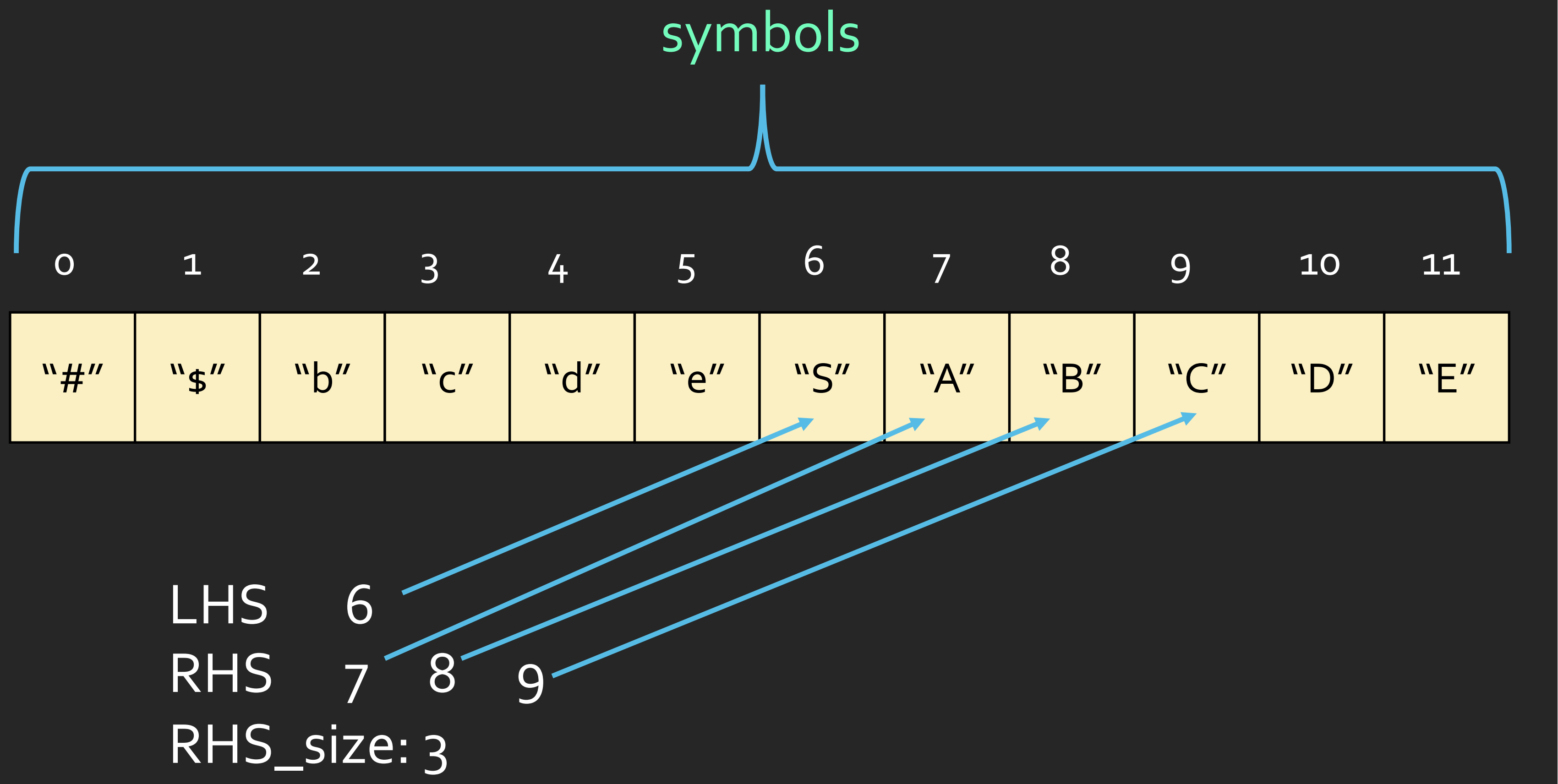
- Let us see how this can be done and then we get back to the various tasks

Note: You are not required to follow the representation that I will describe next and some students do not find it to be the easiest to work with

Grammar Representation with indices

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)

Rule 1



symbols

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
RHS_size: 3

Grammar Representation with indices

S → A B C (1)

A → D E (2)

B → b B (3)

B → ε (4)

C → c C (5)

C → ε (6)

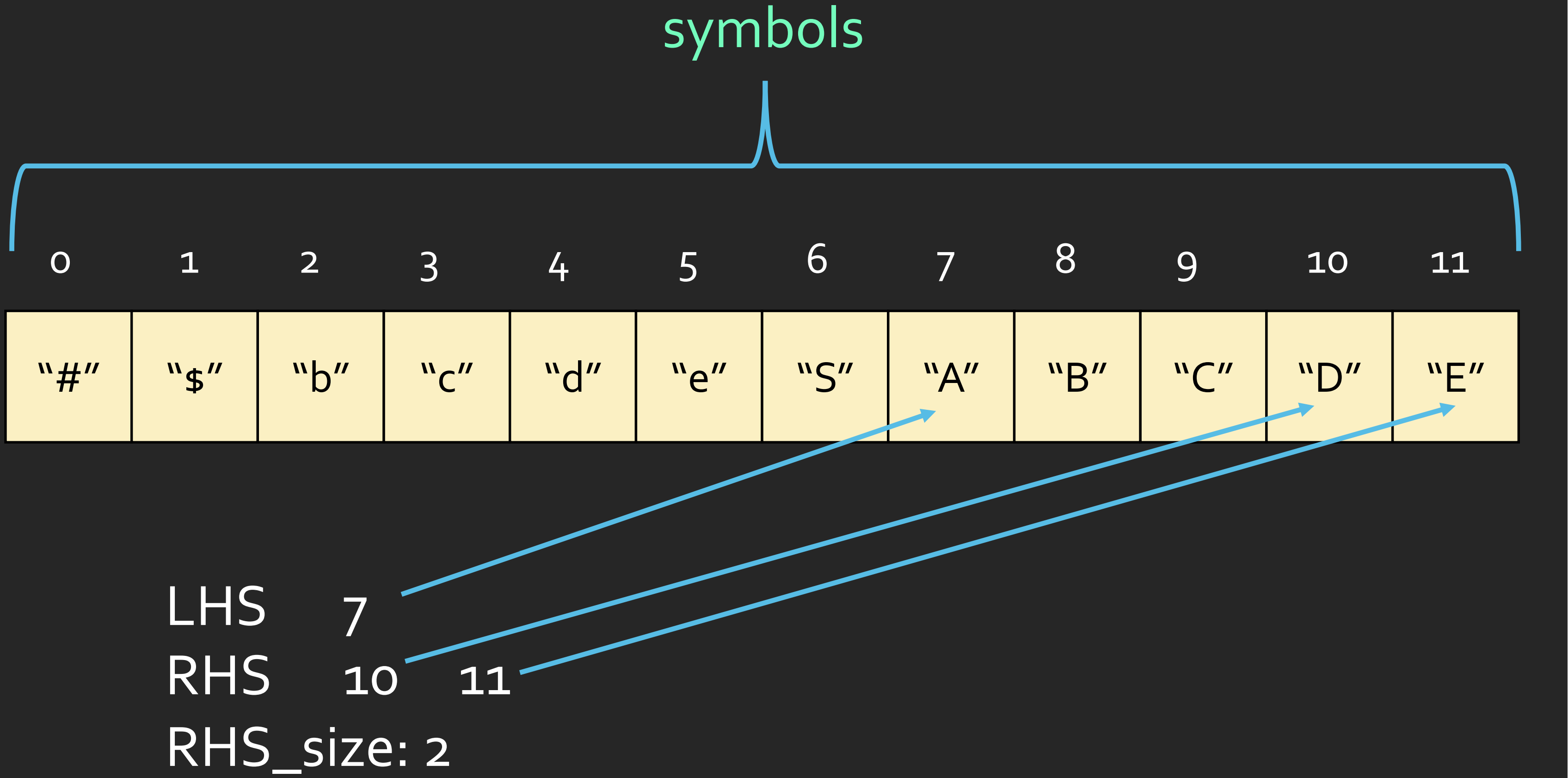
D → d D (7)

D → ε (8)

E → e E (9)

E → ε (10)

Rule 2



symbols

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

LHS 7

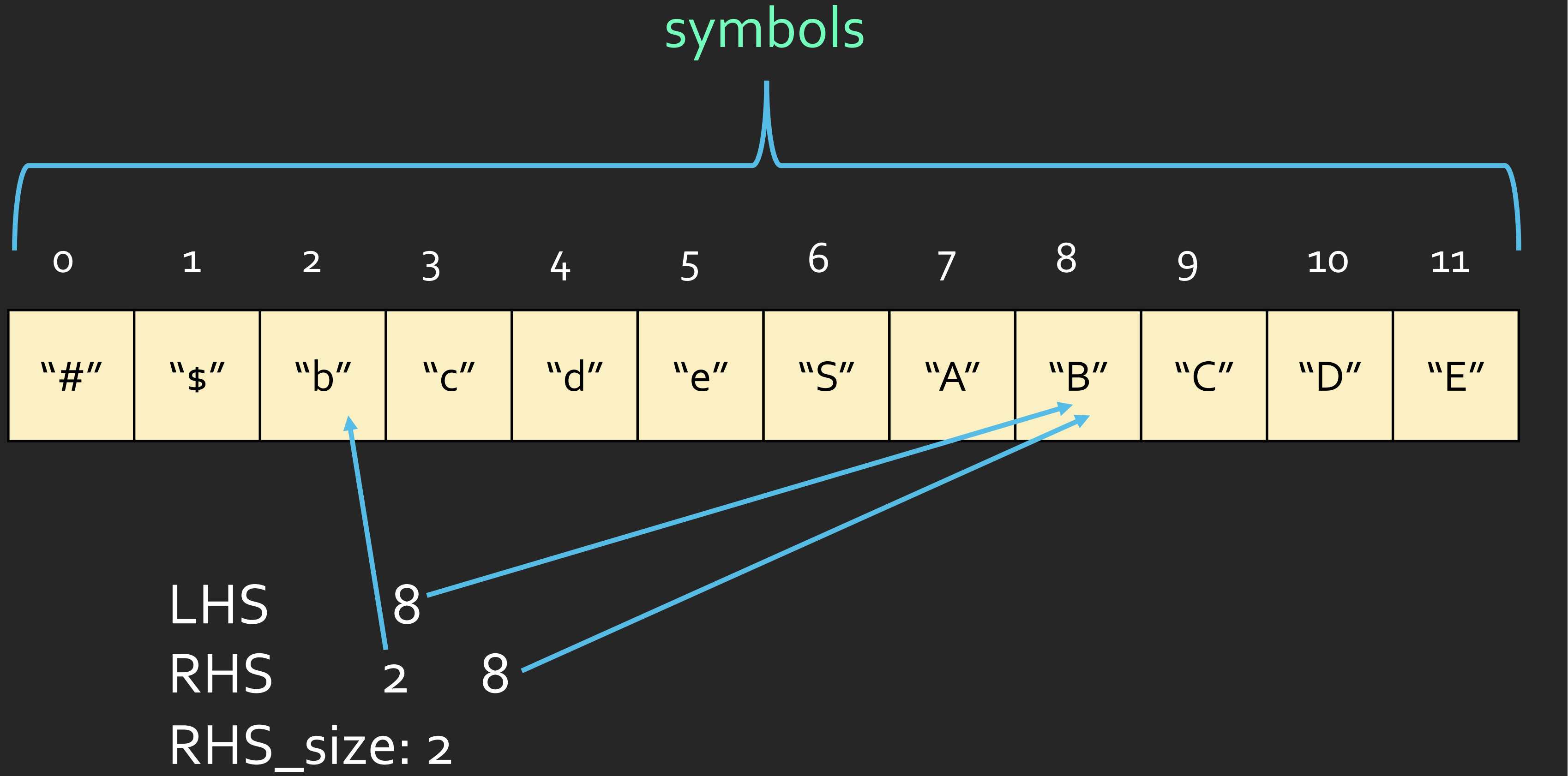
RHS 10 11

RHS_size: 2

Grammar Representation with indices

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)

Rule 2



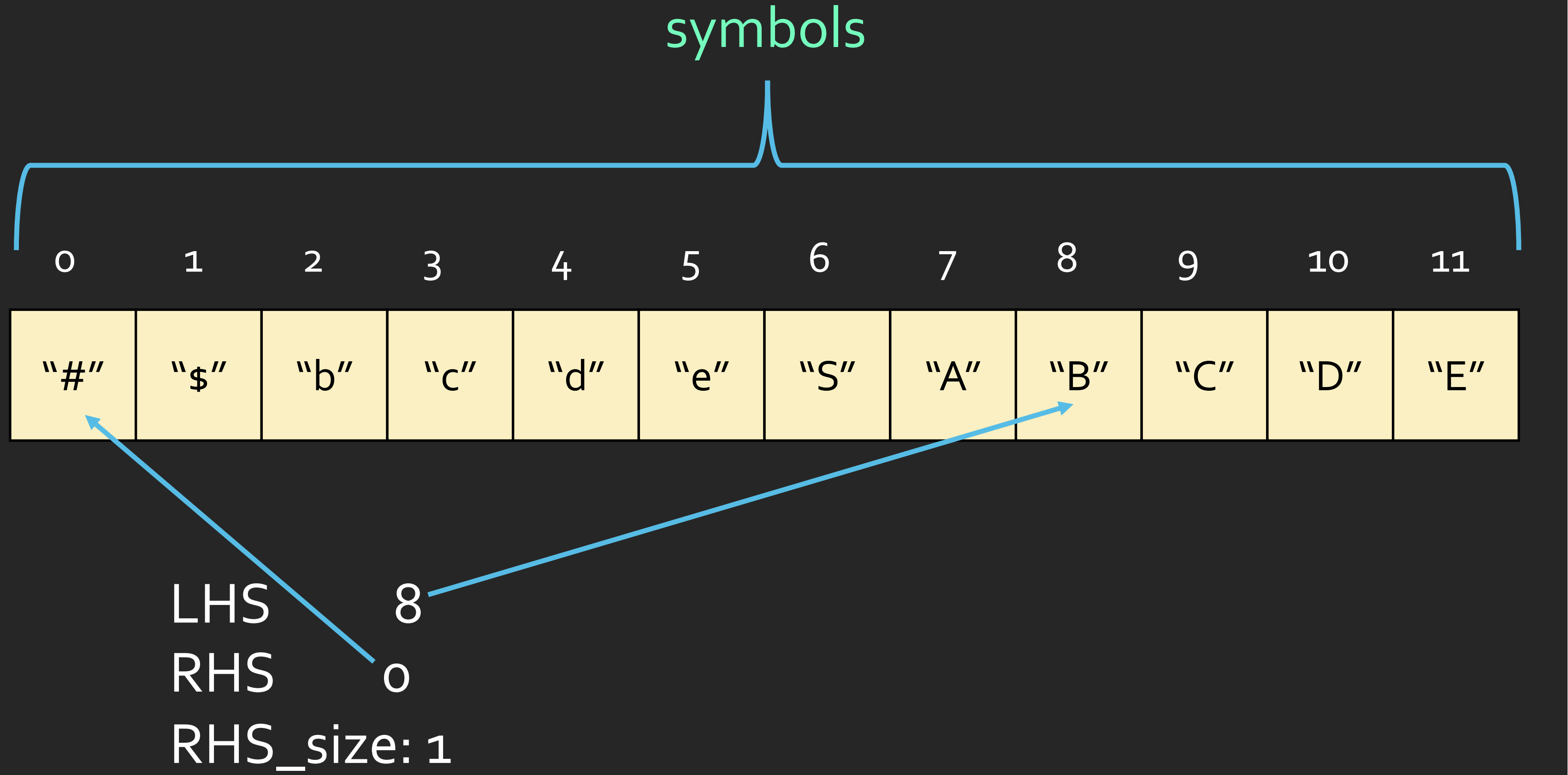
symbols

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

LHS 8
RHS 2 8
RHS_size: 2

Grammar Representation with indices

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 with indices Example

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)

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 with indices

- 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_{index}, you simply print Symbols[A_{index}].

Task 2: 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

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

Example `A -> D E * A -> C F G * E -> d E * C -> c C * F -> f * #`

In this example, all symbols are useless!

Calculating Useless Symbols

The following is the general approach. I will describe steps 1 and 2 in details afterwards

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

Examples of generating symbols

Example

```
A -> D E *
A -> C F G *
D -> d C *
C -> c *
E -> c E
F -> f * #
```

In this example, G, d, c and f are terminals, so G, d, c, F, C, D and A are generating because

```
G =>* G
d =>* d
c =>* c
f =>* f
F =>* f
A =>* cfG
D =>* dc
```

All other symbols are not generating

Examples of reachable symbols

Example

```
A -> D E *
A -> C F G *
D -> d C *
C -> c *
E -> c E
F -> f * #
```

In this example all symbols are reachable because

```
A => D E => d C => d c
A => C F G => C f G
```

Examples of reachable symbols

Example

```
A -> D E *
A -> C F G *
D -> d C *
C -> c *
E -> c E
F -> f * #
```

In this example E and D are useless. All other symbols are useful. Note how D is useless even though it is generating

Step 1. 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

$\Rightarrow$ A is generating

Iterative approach to calculating generating symbols

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"

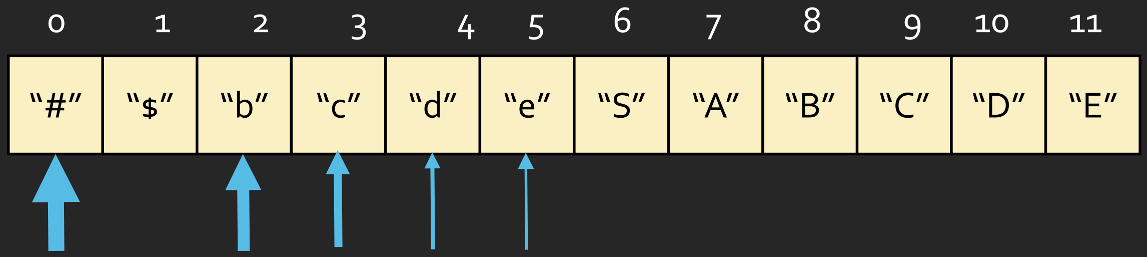
Generating array

0	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)

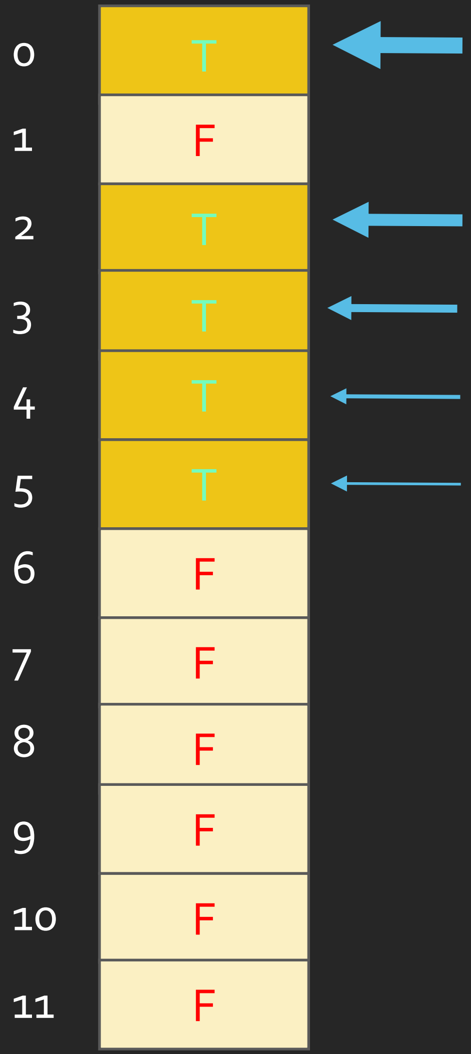
Symbols



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

all terminals are 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"

- We consider one rule at a time starting with the first rule
- 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 → 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 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 → 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"

- 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 → 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’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 → 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"

- 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 → 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”

- 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 → 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 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:

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

Step 2. Calculating reachable symbols

1. S is reachable

2. If $A \to A_1\,A_2 \dots A_k$ is a grammar rule and A is reachable ⇒ 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 from RulesGen that have a non-reachable symbol
- The remaining rules, if any, are all non-useless and you should print them

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 and design to give you feedback. The earlier you ask the better off you will be.