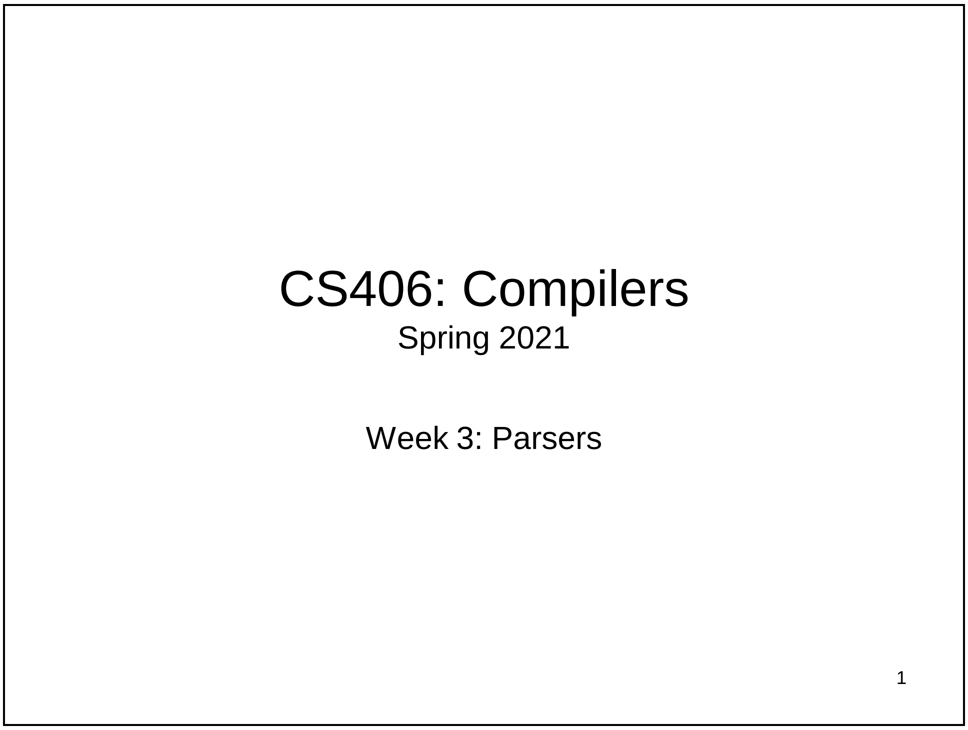


CS406: Compilers

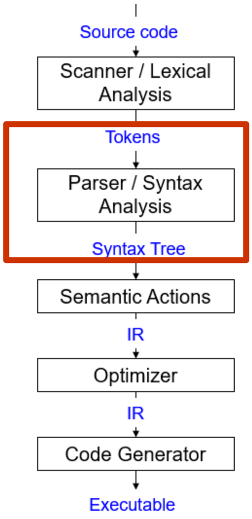
Spring 2021

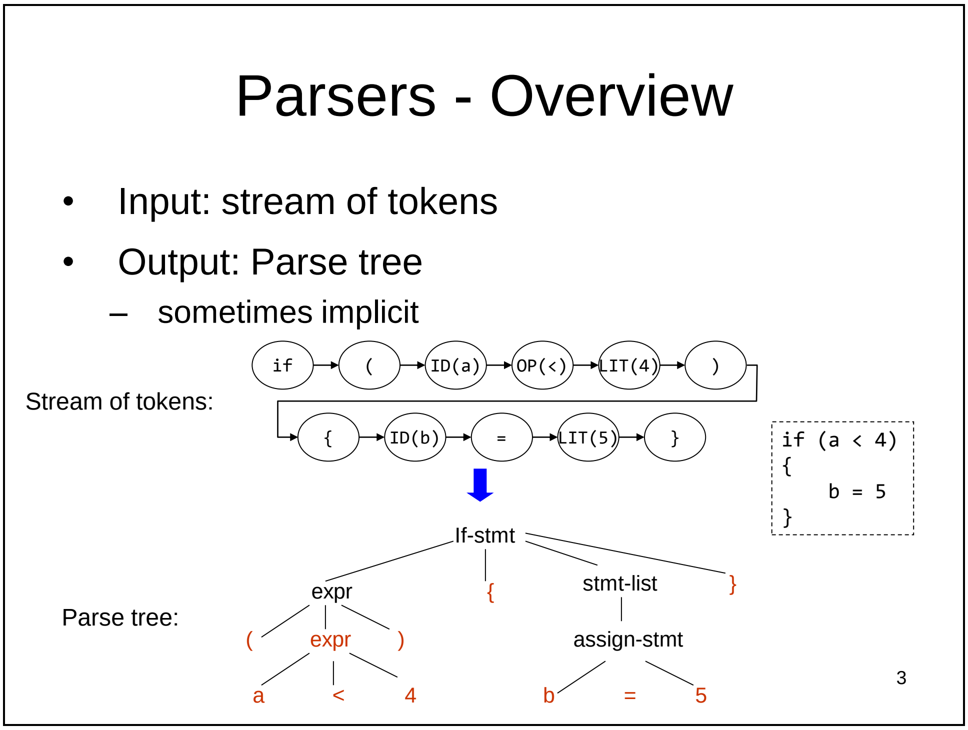
Week 3: Parsers

1

Parsers - Overview

- Also called syntax analyzers
- Determine two things:
  1. Is a program syntactically valid?
     - Is an English sentence grammatically correct?
  2. What is the structure of programming language constructs? E.g. does the sequence `IF, ID(a), OP(<), ID(b), {, ID(a), ASSIGN, LIT(5), }, ;, }` refer to an `if-statement`?
     - Diagramming English sentences





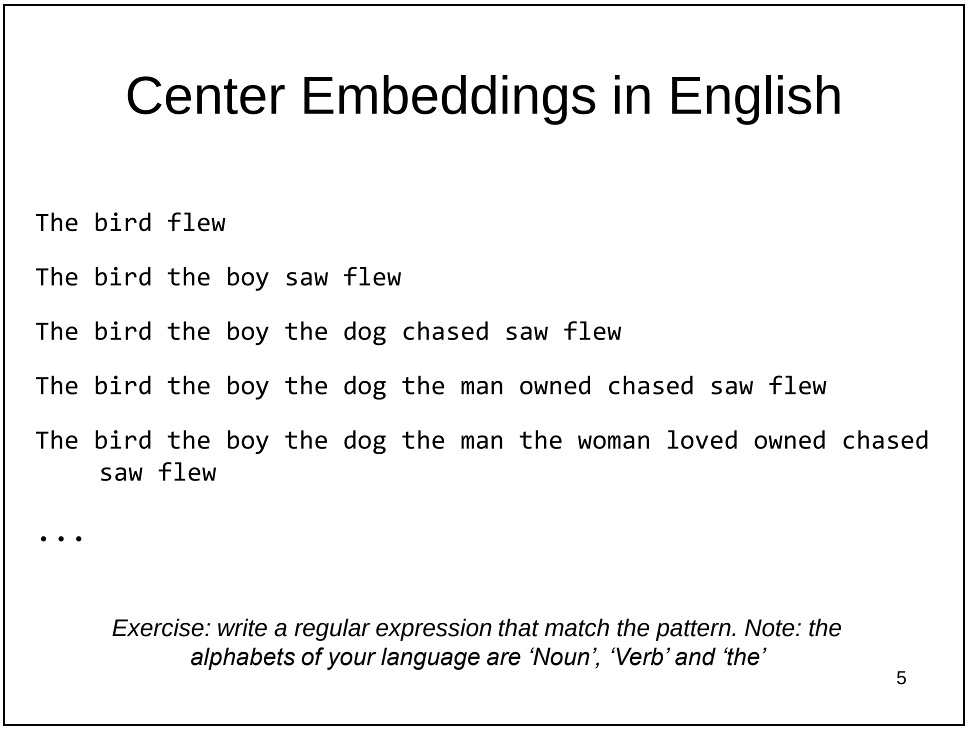
Parsers - Overview

- Input: stream of tokens
- Output: Parse tree
  - sometimes implicit

```
if (a < 4)
{
    b = 5
}
```

Parsers – what do we need to know?

1. How do we define language constructs?
   - Context-free grammars
2. How do we determine: 1) valid strings in the language? 2) structure of program?
   - LL Parsers, LR Parsers
3. How do we write Parsers?
   - E.g. use a parser generator tool such as Bison

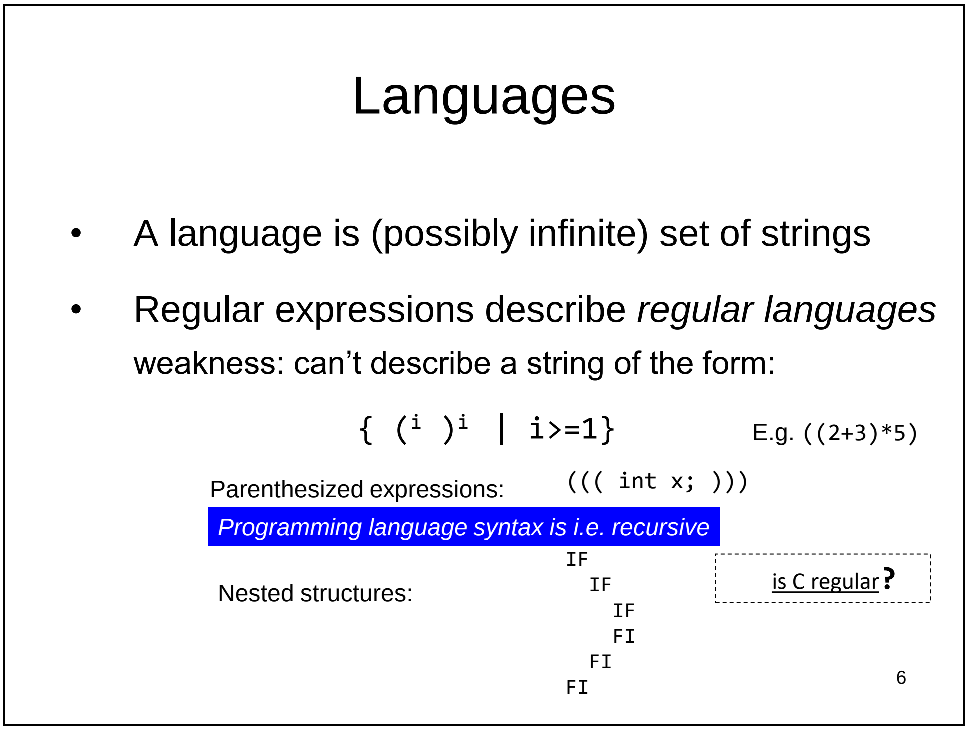


Center Embeddings in English

```
The bird flew

The bird the boy saw flew

The bird the boy the dog chased saw flew

The bird the boy the dog the man owned chased saw flew

The bird the boy the dog the man the woman loved owned chased
    saw flew

...
```

Exercise: write a regular expression that match the pattern. Note: the alphabets of your language are ‘Noun’, ‘Verb’ and ‘the’

You can construct arbitrarily long sentences like this in English.



Languages

- A language is (possibly infinite) set of strings
- Regular expressions describe *regular languages*

  weakness: can’t describe a string of the form:

$\{ (^{i}\ )^{i} \mid i>=1\}$ E.g. ((2+3)*5)

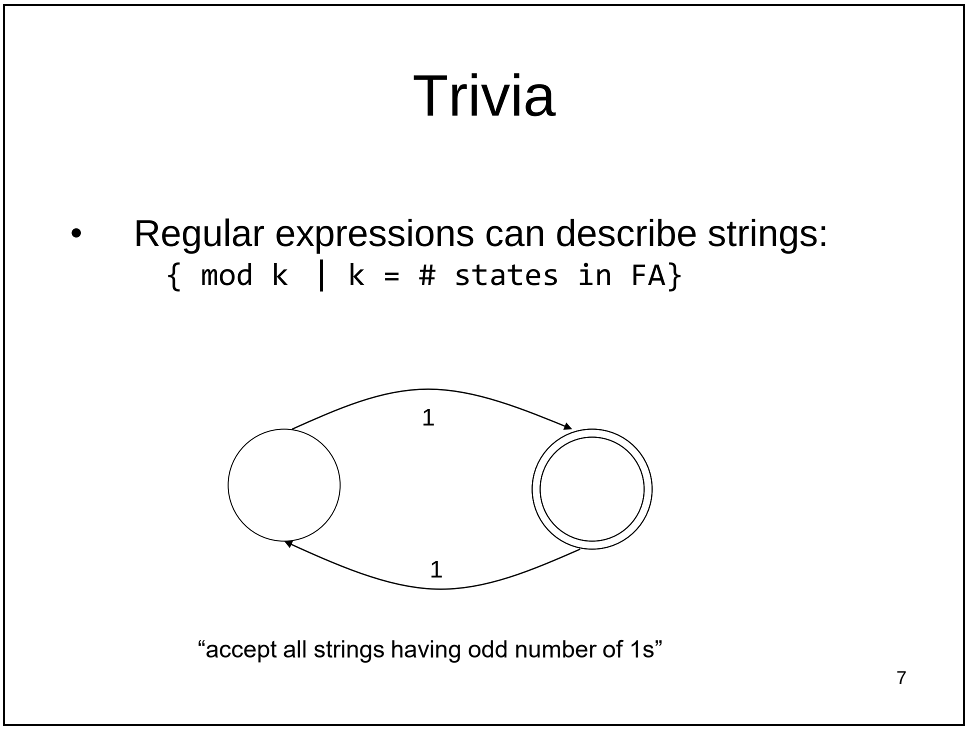
Parenthesized expressions: `((( int x; )))`

Programming language syntax is i.e. recursive

Nested structures:

```
IF
  IF
    IF
    FI
  FI
FI
```

is C regular?



Trivia

- Regular expressions can describe strings:
  `{ mod k | k = # states in FA}`

“accept all strings having odd number of 1s”

7

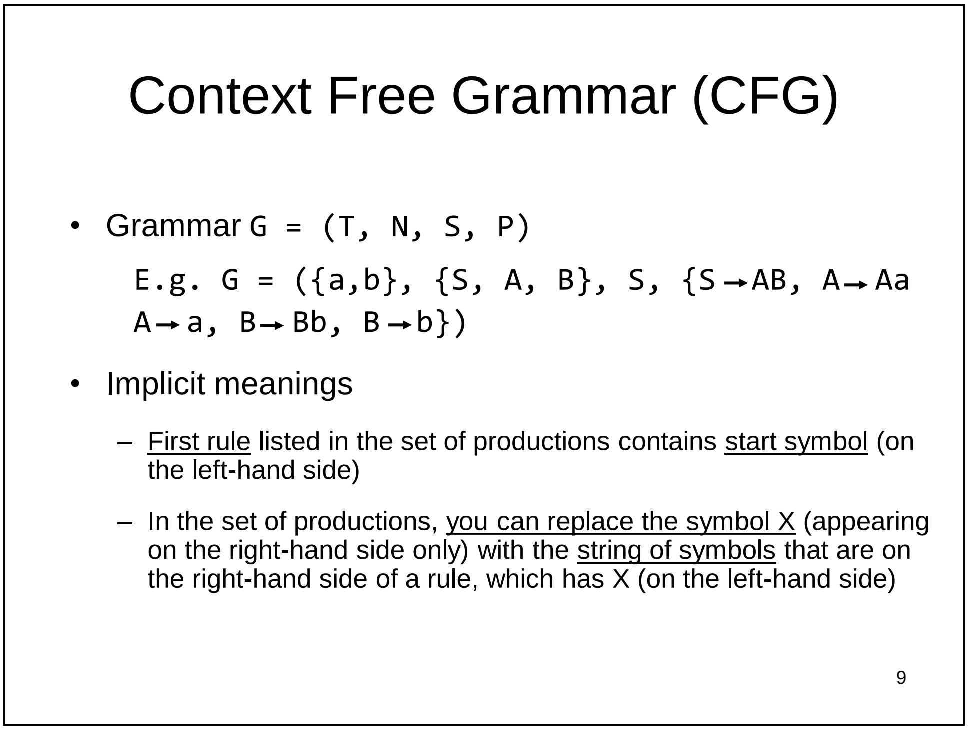
What FAs and regular expressions can do is describe strings of the form “odd number of 1s”, they can determine parity but cannot count.

Context Free Grammar (CFG)

- Natural notation for describing recursive structure definitions. Hence, suitable for specifying language constructs.

- Consist of:
  - A set of *Terminals* (T)
  - A set of *Non-terminals* (N)
  - A *Start Symbol* ($S \in N$)
  - A *set of Productions* ($X \texttt{ -> } Y_1..Y_N$) (aka. rules)

$$P: X \longrightarrow Y_1 Y_2 Y_3 .. Y_N \mid X \in N,\ Y_i \in N \cup T \cup \epsilon/\lambda$$

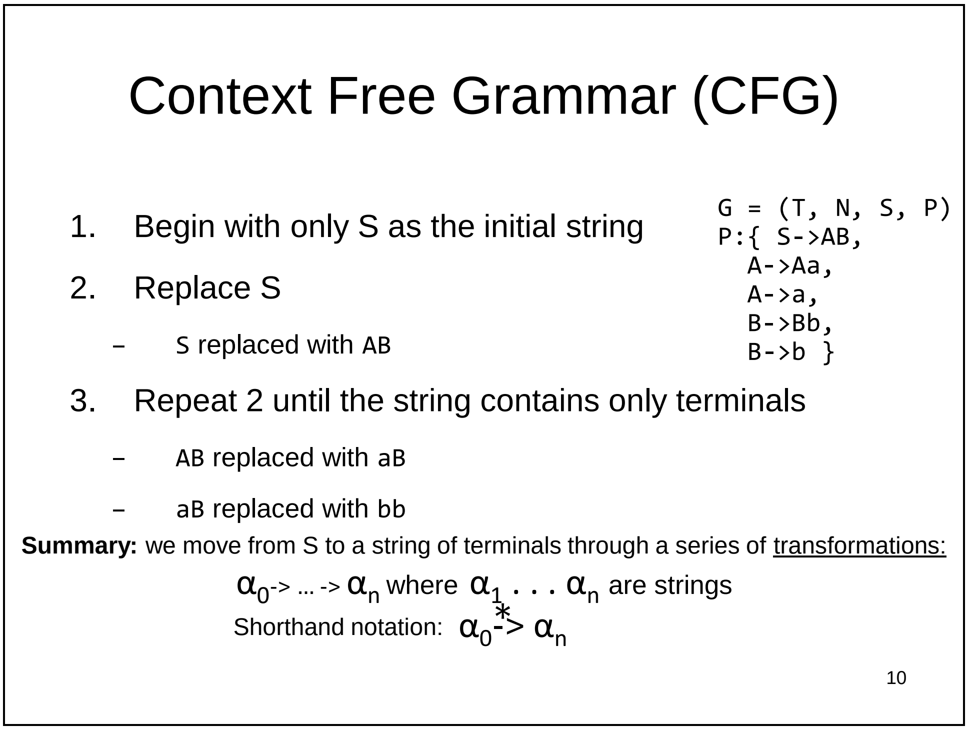


Context Free Grammar (CFG)

- Grammar `G = (T, N, S, P)`

  `E.g. G = ({a,b}, {S, A, B}, S, {S → AB, A → Aa`
  `A → a, B → Bb, B → b})`

- Implicit meanings
  - First rule listed in the set of productions contains start symbol (on the left-hand side)
  - In the set of productions, you can replace the symbol X (appearing on the right-hand side only) with the string of symbols that are on the right-hand side of a rule, which has X (on the left-hand side)



Context Free Grammar (CFG)

```
G = (T, N, S, P)
P:{ S->AB,
  A->Aa,
  A->a,
  B->Bb,
  B->b }
```

1. Begin with only S as the initial string
2. Replace S
   - S replaced with AB
3. Repeat 2 until the string contains only terminals
   - AB replaced with aB
   - aB replaced with bb

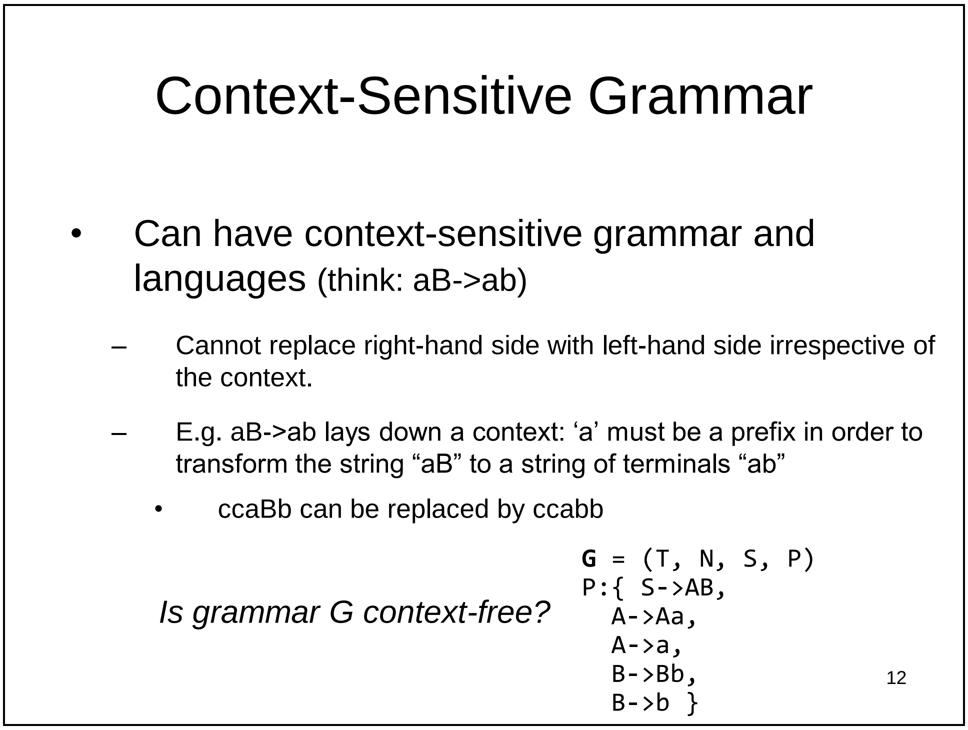
Summary: we move from S to a string of terminals through a series of transformations:

α_0-> ... -> α_n where $\alpha_1 \ldots \alpha_n$ are strings

Shorthand notation: $\alpha_0 \overset{*}{->} \alpha_n$

Language of the Grammar

- Language L(G) of the context-free grammar G
  - Set of strings that can be derived from S
  - $\{a_1a_2a_3..a_N \mid a_i \in T \; \forall i \text{ and } S \overset{*}{->} a_1a_2a_3..a_N \}$
  - Is called context-free language
    - All regular languages are context-free but not vice-versa.
    - Can have many grammars generating same language.

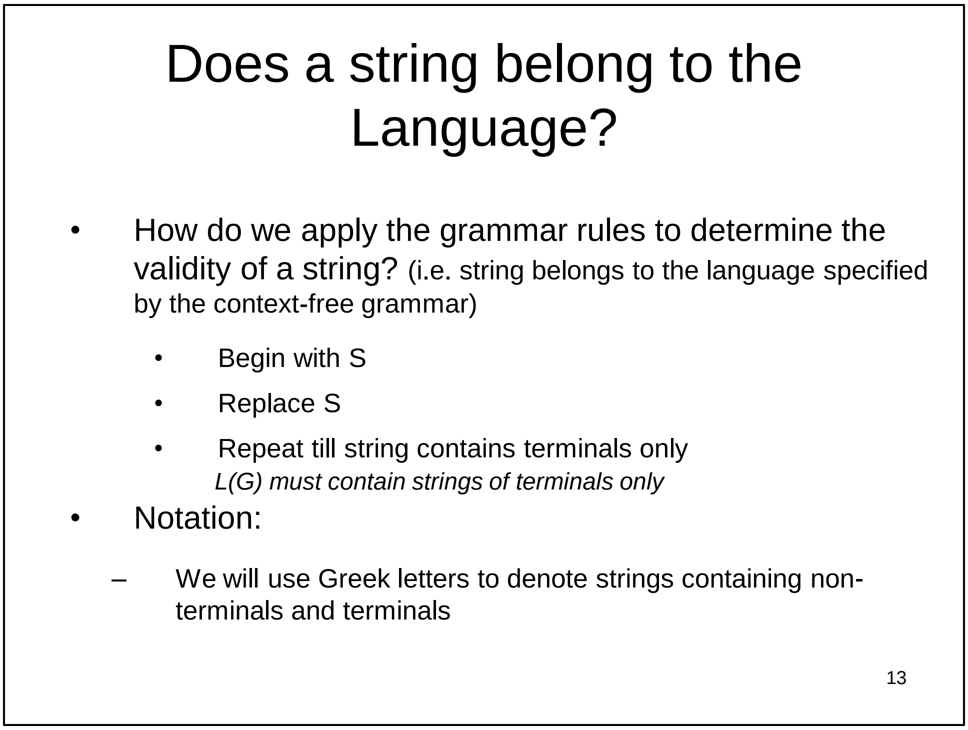


Context-Sensitive Grammar

- Can have context-sensitive grammar and languages (think: aB->ab)
  - Cannot replace right-hand side with left-hand side irrespective of the context.
  - E.g. aB->ab lays down a context: ‘a’ must be a prefix in order to transform the string “aB” to a string of terminals “ab”
    - ccaBb can be replaced by ccabb

Is grammar G context-free?

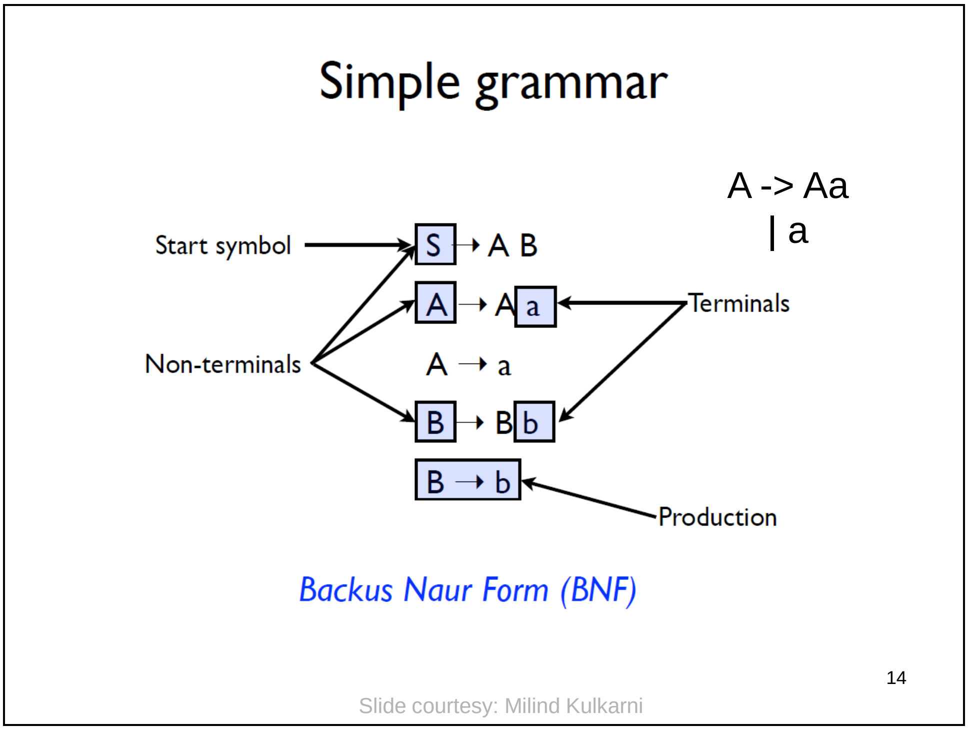
```
G = (T, N, S, P)
P:{ S->AB,
  A->Aa,
  A->a,
  B->Bb,
  B->b }
```



Does a string belong to the Language?

- How do we apply the grammar rules to determine the validity of a string? (i.e. string belongs to the language specified by the context-free grammar)
  - Begin with S
  - Replace S
  - Repeat till string contains terminals only

    L(G) must contain strings of terminals only
- Notation:
  - We will use Greek letters to denote strings containing non-terminals and terminals



Simple grammar

A -> Aa
| a

Start symbol

Non-terminals

S → A B

A → A a

A → a

B → B b

B → b

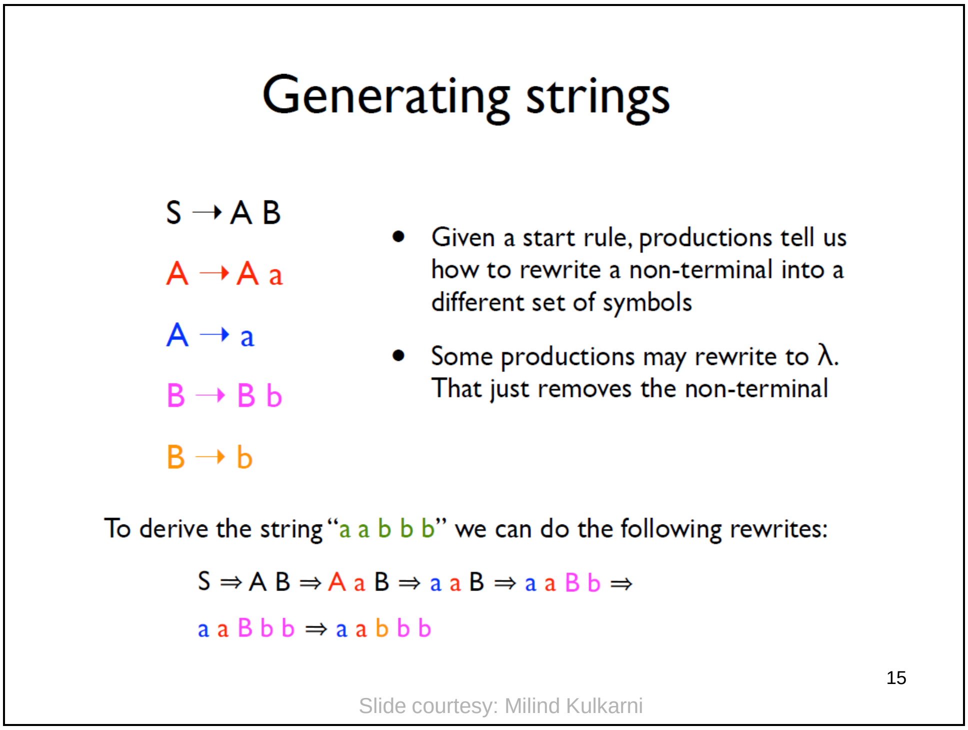
Terminals

Production

Backus Naur Form (BNF)

14

Slide courtesy: Milind Kulkarni



Generating strings

S → A B

A → A a

A → a

B → B b

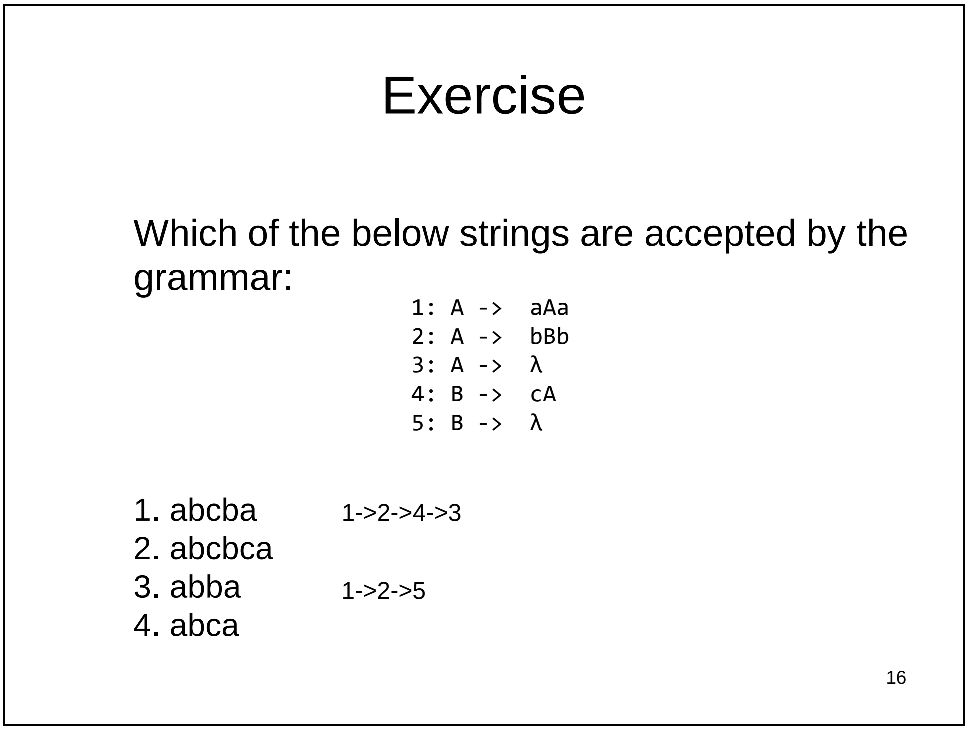
B → b

- Given a start rule, productions tell us how to rewrite a non-terminal into a different set of symbols
- Some productions may rewrite to λ. That just removes the non-terminal

To derive the string "a a b b b" we can do the following rewrites:

S ⇒ A B ⇒ A a B ⇒ a a B ⇒ a a B b ⇒
a a B b b ⇒ a a b b b

Slide courtesy: Milind Kulkarni

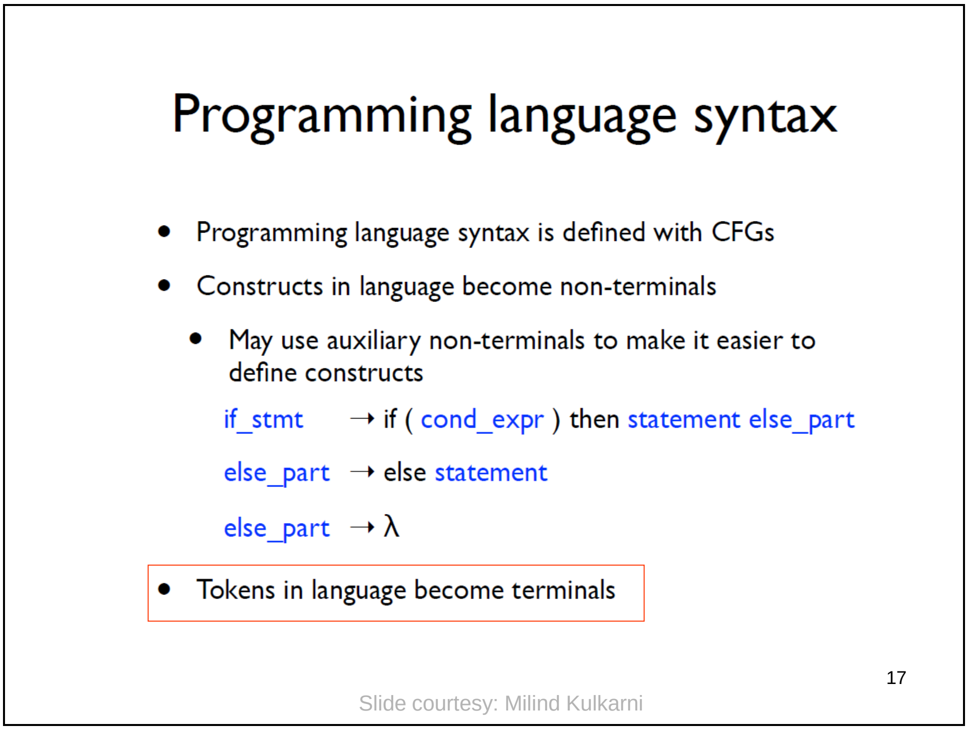


Exercise

Which of the below strings are accepted by the grammar:

```
1: A ->  aAa
2: A ->  bBb
3: A ->  λ
4: B ->  cA
5: B ->  λ
```

1. abcba 1->2->4->3
2. abcbca
3. abba 1->2->5
4. abca



Programming language syntax

- Programming language syntax is defined with CFGs
- Constructs in language become non-terminals
  - May use auxiliary non-terminals to make it easier to define constructs

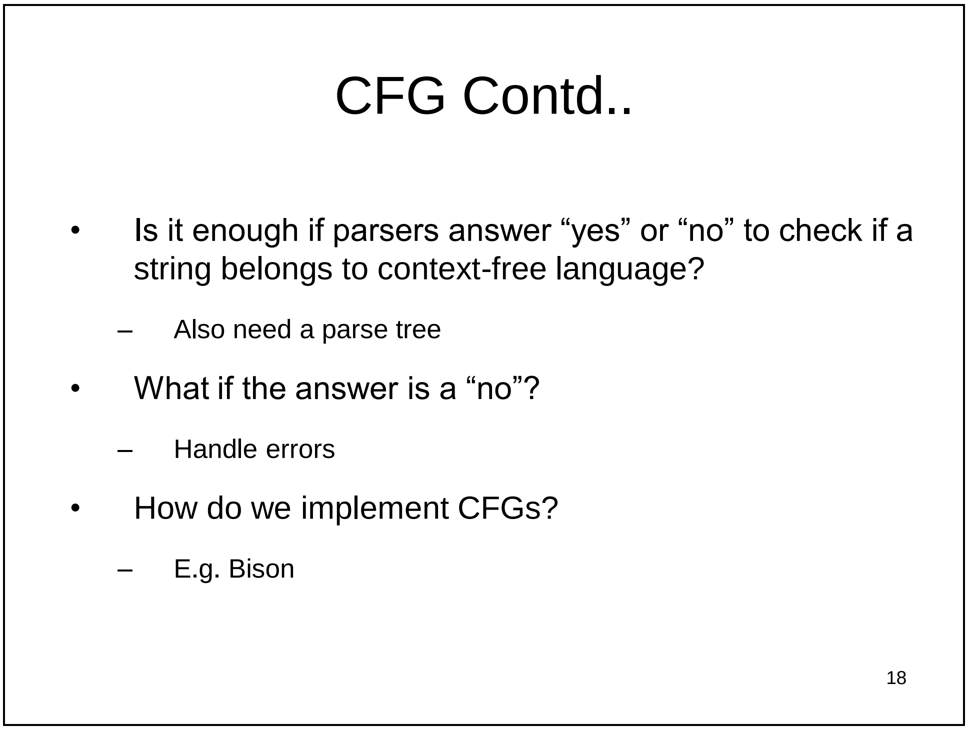
if_stmt → if (cond_expr) then statement else_part

else_part → else statement

else_part → λ

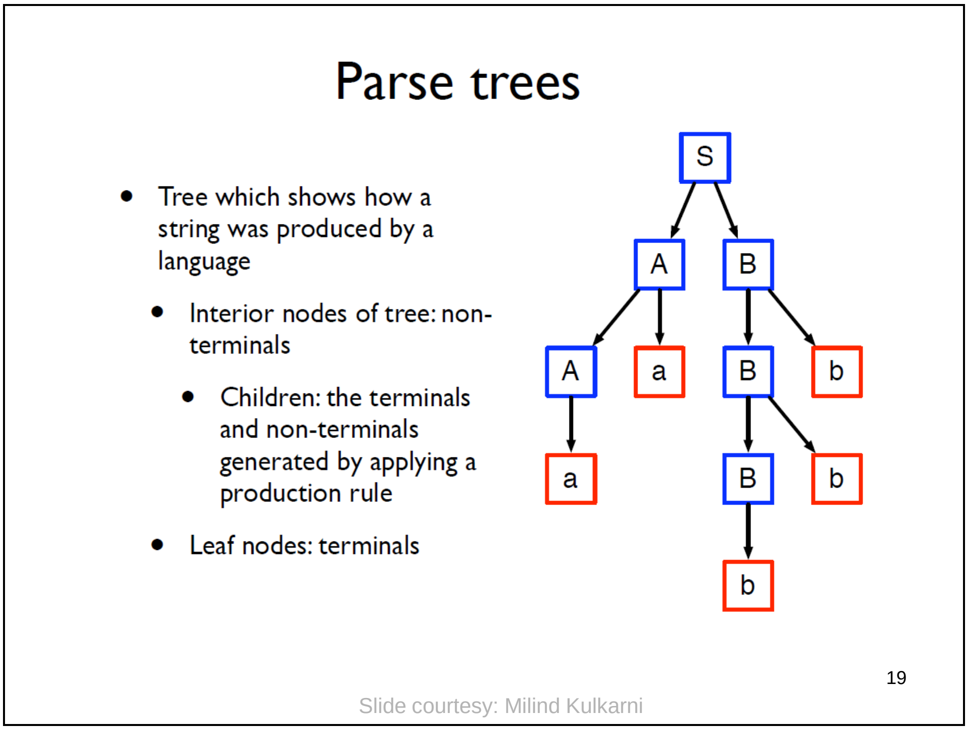
- Tokens in language become terminals

Slide courtesy: Milind Kulkarni



CFG Contd..

- Is it enough if parsers answer “yes” or “no” to check if a string belongs to context-free language?
  - Also need a parse tree
- What if the answer is a “no”?
  - Handle errors
- How do we implement CFGs?
  - E.g. Bison



Parse trees

- Tree which shows how a string was produced by a language
  - Interior nodes of tree: non-terminals
    - Children: the terminals and non-terminals generated by applying a production rule
  - Leaf nodes: terminals

S
A
B
A
a
B
b
a
B
b
b

Slide courtesy: Milind Kulkarni

Parse Trees and String Derivations

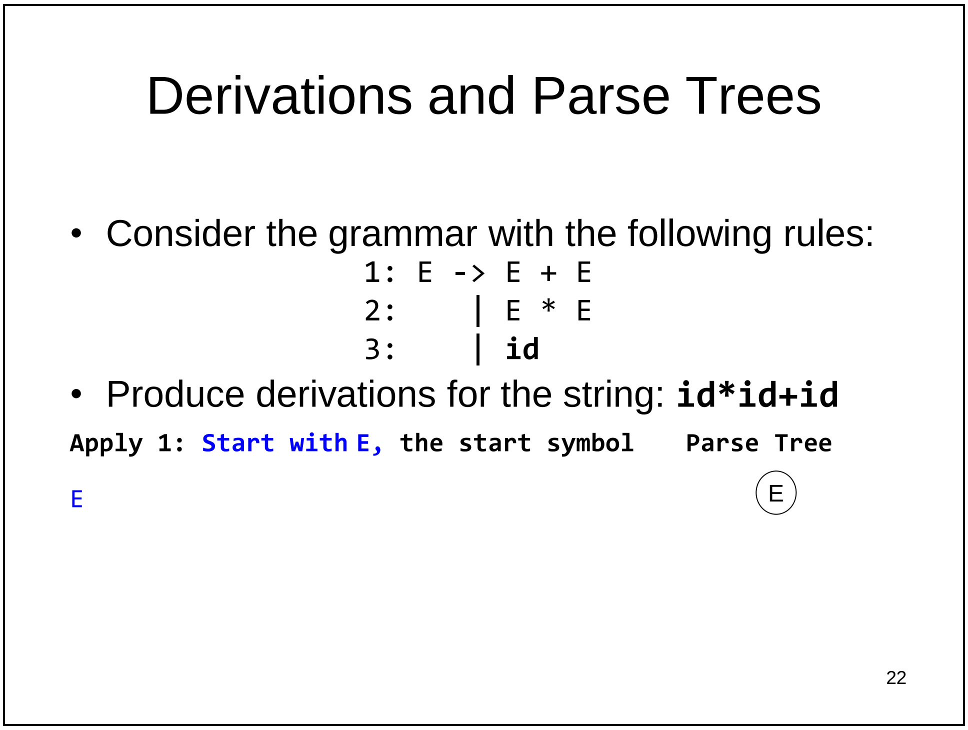
- Recall: sequence of rules applied to produce a string is a derivation
- A derivation defines a parse tree
  - A parse tree may have many derivations

Derivations and Parse Trees

- Consider the grammar with the following rules:

```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **`id*id+id`**



Derivations and Parse Trees

- Consider the grammar with the following rules:

```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **id*id+id**

Apply 1: Start with E, the start symbol **Parse Tree**

E

Derivations and Parse Trees

- Consider the grammar with the following rules:

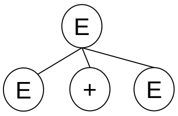
```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **`id*id+id`**

`Apply 1:` **Replace `E` with `E + E`**

`Parse Tree`

```
E
E+E
```



Derivations and Parse Trees

- Consider the grammar with the following rules:

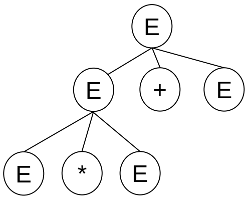
```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **`id*id+id`**

`Apply 2:` **Replace `E` with `E * E`**

`Parse Tree`

```
E
E+E
E*E+E
```



Derivations and Parse Trees

- Consider the grammar with the following rules:

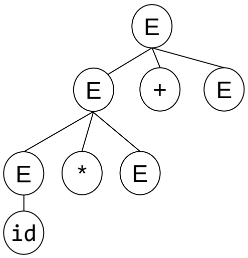
```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **id*id+id**

Apply 3: Replace E with id

```
E
E+E
E*E+E
id*E+E
```

Parse Tree



Derivations and Parse Trees

- Consider the grammar with the following rules:

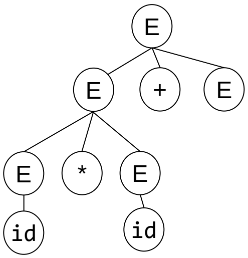
```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **`id*id+id`**

`Apply 3:` **Replace `E` with `id`**

```
E
E+E
E*E+E
id*E+E
id*id+E
```

`Parse Tree`



Derivations and Parse Trees

- Consider the grammar with the following rules:

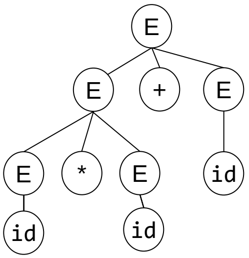
```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **id*id+id**

Apply 3: Replace E with id

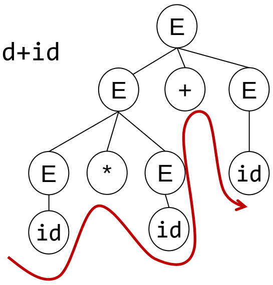
```
E
E+E
E*E+E
id*E+E
id*id+E
id*id+id
```

Parse Tree



Derivations and Parse Trees

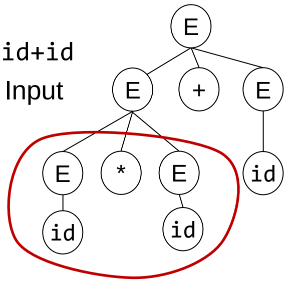
- Note in previous slides:
  - Replacement done on left-most non-terminal in the string - **called left-most derivation**
  - Terminals at leaves and non-terminal as interior nodes
  - `Inorder` traversal produces input string `id*id+id`



Derivations and Parse Trees

- Note in previous slides:
  - Replacement done on left-most non-terminal in the string - **called left-most derivation**
  - Terminals at leaves and non-terminal as interior nodes
  - `Inorder` traversal produces input string `id*id+id`
  - Parse tree shows association of operations. Input string doesn't
    - `*` associated with identifiers in the subtree

`(id * id)+id`



Derivations and Parse Trees

- Consider the grammar with the following rules:

```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **`id*id+id`**
  - Using right-most derivations

    i.e. replace the right-most non-terminal

Derivations and Parse Trees

- Consider the grammar with the following rules:

```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **id*id+id**

Start with E, the start symbol

E

Parse Tree

Derivations and Parse Trees

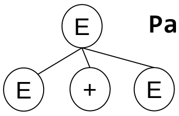
- Consider the grammar with the following rules:

```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **id*id+id**

Apply 2: Replace E with E+E

```
E
E+E
```



Parse Tree

Derivations and Parse Trees

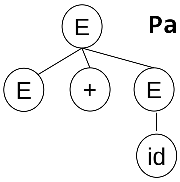
- Consider the grammar with the following rules:

```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **`id*id+id`**

`Apply 1:` **Replace `E` with id**

```
E
E+E
E+id
```



`Parse Tree`

Derivations and Parse Trees

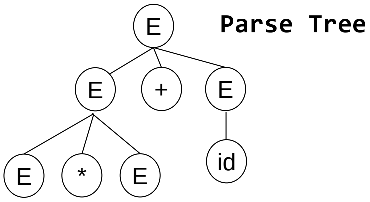
- Consider the grammar with the following rules:

```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **`id*id+id`**

`Apply 3:` **Replace `E` with E * E**

```
E
E+E
E+id
E*E+id
```



Derivations and Parse Trees

- Consider the grammar with the following rules:

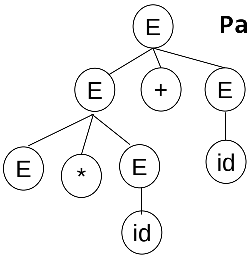
```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **`id*id+id`**

`Apply 3:` **Replace `E` with id**

```
E
E+E
E+id
E*E+id
E*id+id
```

`Parse Tree`



35

Derivations and Parse Trees

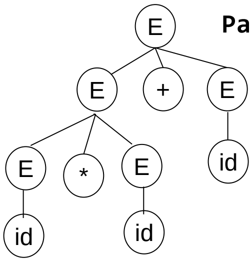
- Consider the grammar with the following rules:

```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **`id*id+id`**

`Apply 3:` **Replace `E` with id**

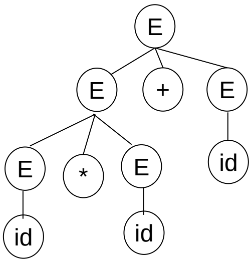
```
E
E+E
E+id
E*E+id
E*id+id
id*id+id
```



Parse Tree

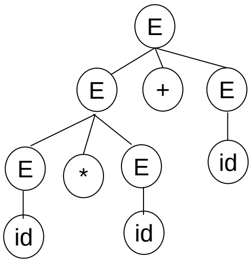
Derivations and Parse Trees

- We get the same parse tree using left-most and right-most derivations.
  - Every parse tree has left-most and right-most (and any random order) derivations.



Derivations and Parse Trees

- We get the same parse tree using left-most and right-most derivations.
  - Every parse tree has left-most and right-most (and any random order) derivations.



- But there could be a string (or more than one strings) for which there exists derivations that would get different parse trees

Derivations and Parse Trees

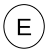
- Consider the grammar with the following rules:

```
1: E -> E + E
2:    | E * E
3:    | id
```

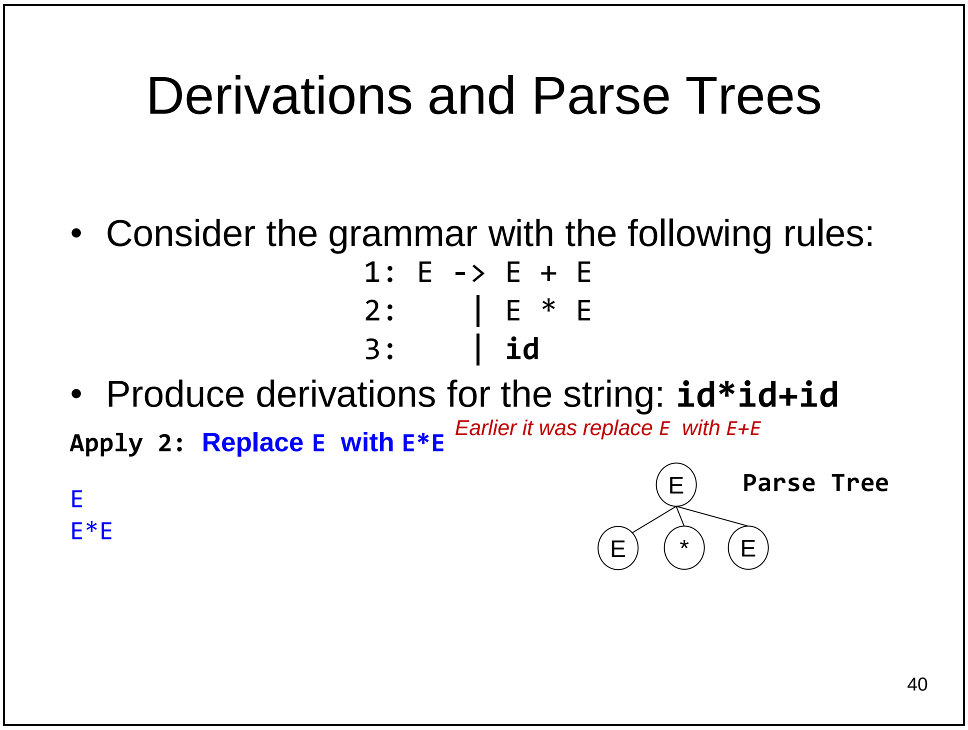
- Produce derivations for the string: **id*id+id**

Start with E, the start symbol

E



Parse Tree



Derivations and Parse Trees

- Consider the grammar with the following rules:

```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **`id*id+id`**

`Apply 2:` **Replace `E` with `E*E`** *Earlier it was replace `E` with `E+E`*

```
E
E*E
```

`Parse Tree`

Derivations and Parse Trees

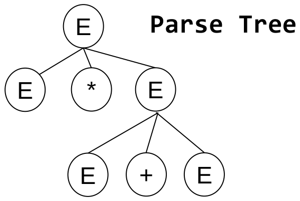
- Consider the grammar with the following rules:

```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **`id*id+id`**

`Apply 1:` **Replace `E` with `E+E`**

```
E
E*E
E*E+E
```



Derivations and Parse Trees

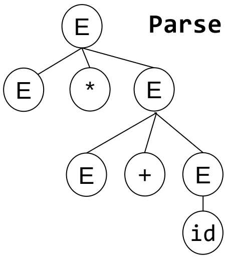
- Consider the grammar with the following rules:

```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **`id*id+id`**

`Apply 3:` Replace `E` with `id`

```
E
E*E
E*E+E
E*E+id
```



Parse Tree

Derivations and Parse Trees

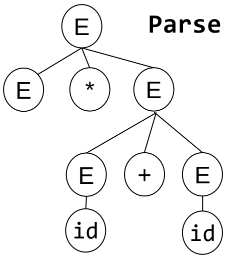
- Consider the grammar with the following rules:

```
1: E -> E + E
2:    | E * E
3:    | id
```

- Produce derivations for the string: **`id*id+id`**

`Apply 3:` **Replace `E` with `id`**

```
E
E*E
E*E+E
E*E+id
E*id+id
```



`Parse Tree`

Derivations and Parse Trees

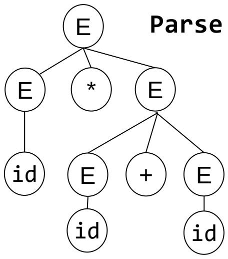
- Consider the grammar with the following rules:

```
1: E -> E + E
2:    | E * E
3:    | id
```

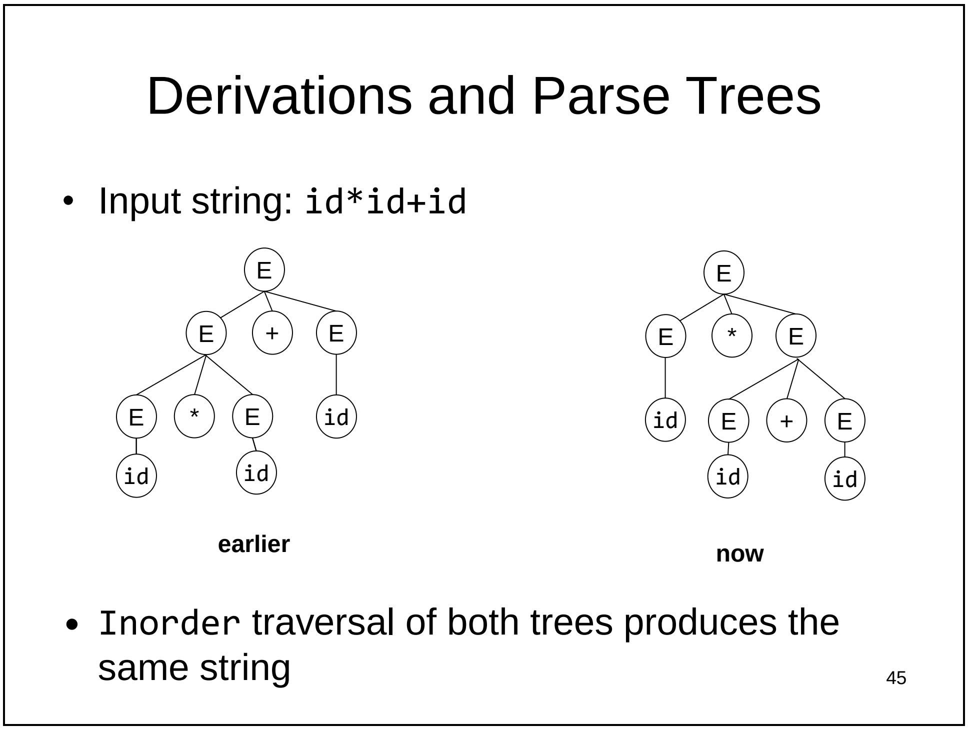
- Produce derivations for the string: **id*id+id**

Apply 3: Replace E with id

```
E
E*E
E*E+E
E*E+id
E*id+id
id*id+id
```



Parse Tree



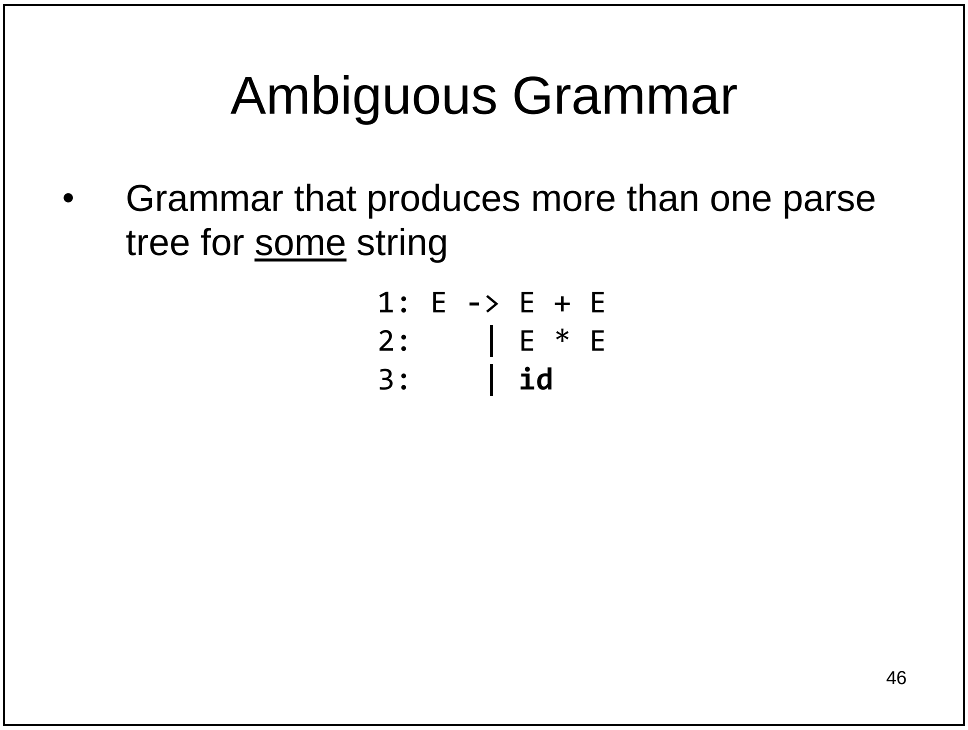
Derivations and Parse Trees

- Input string: `id*id+id`

earlier

now

- `Inorder` traversal of both trees produces the same string



Ambiguous Grammar

- Grammar that produces more than one parse tree for some string

```
1: E -> E + E
2:      | E * E
3:      | id
```

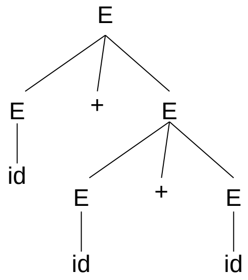
Ambiguity – what to do?

- Ignore it (let it be ambiguous)
  - Give hints to other components of the compiler on how to resolve it
- Fix it
  - Manually
  - May make the grammar complicated and difficult to maintain

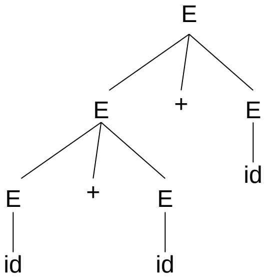
Ambiguity – ignore

- `E -> E + E | id`

```
E->E+E
E->id+E
E->id+E+E
E->id+id+E
E->id+id+id

Produces:
id+(id+id)
```



```
E->E+E
E->E+E+E
E->id+E+E
E->id+id+E
E->id+id+id

Produces:
(id+id)+id
```



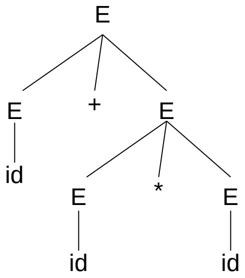
- Associativity declaration in Bison:
  `%left +` Picks the parse tree on the right

Ambiguity - ignore

- **E -> E + E | E * E | id**

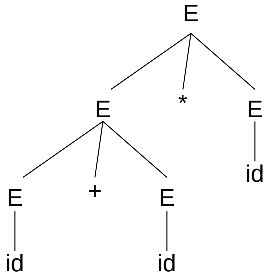
```
E->E+E
E->id+E
E->id+E*E
E->id+id*E
E->id+id*id
```

Produces:
id+(id*id)



```
E->E*E
E->E+E*E
E->id+E*E
E->id+id*E
E->id+id*id
```

Produces:
(id+id)*id



%left +
%left *

*Tells that * has higher precedence over + and both are left associative. So, we get the tree on left.*

Ambiguity – fixing

- Rewrite

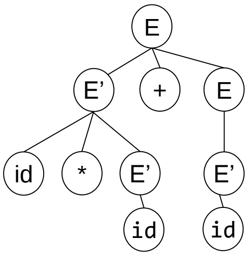
```
E -> E + E
   | E * E
   | id
```

as:

```
E -> E’ + E | E’
E’ -> id * E’ | id
   | (E) * E’ | (E)
```

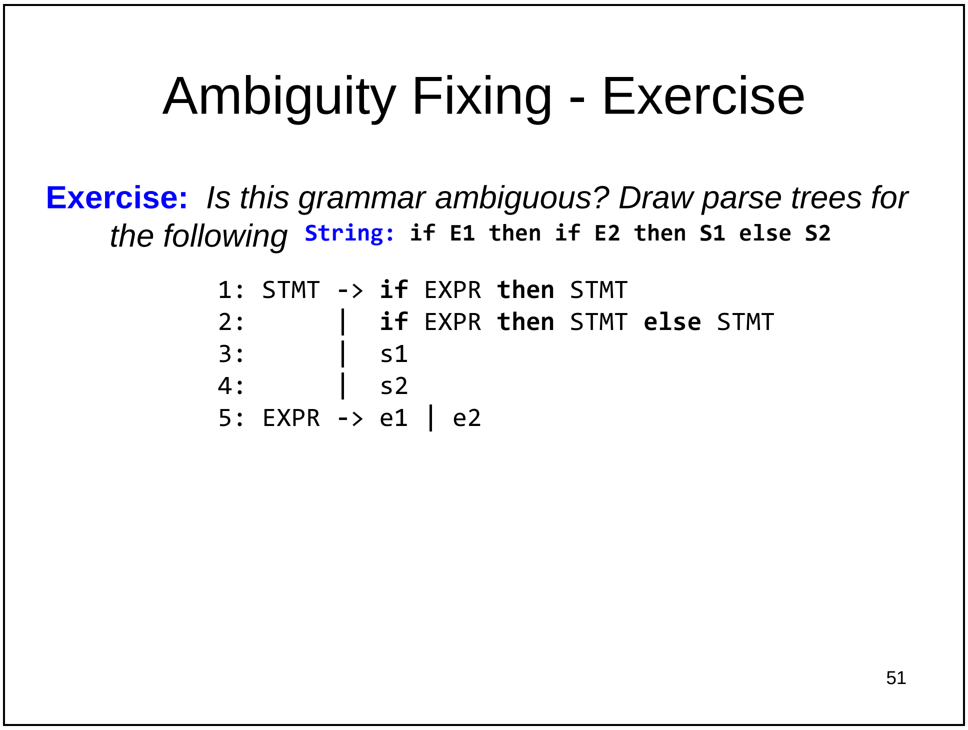
```
E->E’+E
E’->id*E’
E’->id
E->E’
E’->id
```

Is the above sequence left-most or right-most derivation?



E controls generation of +

E’ controls generation of *. *’s are nested deeper in the parse tree.



Ambiguity Fixing - Exercise

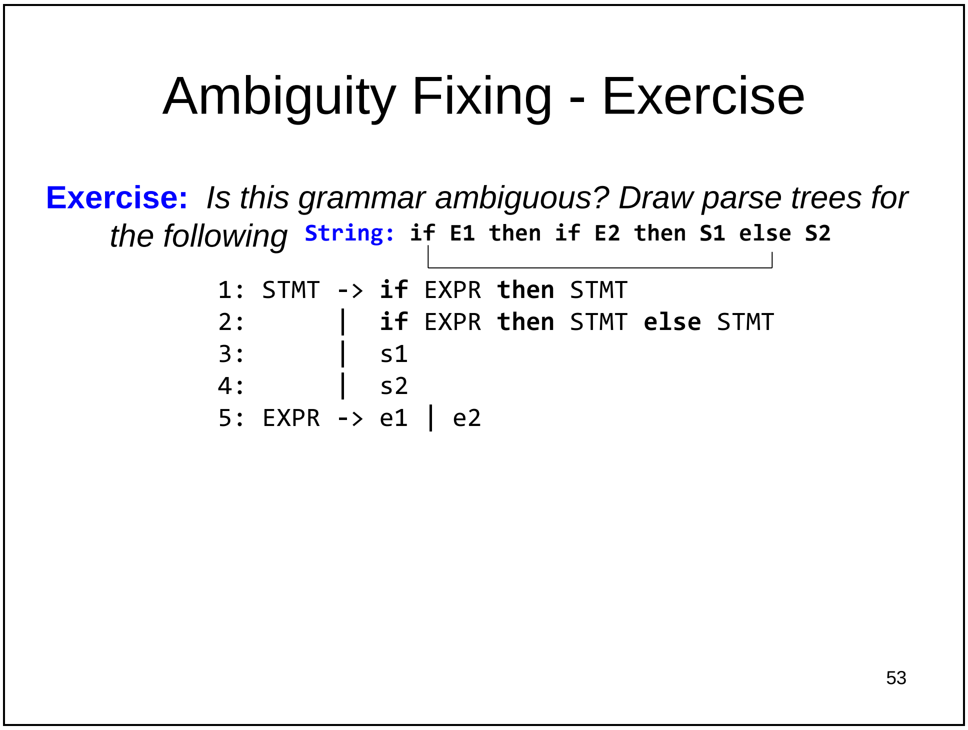
Exercise: *Is this grammar ambiguous? Draw parse trees for the following* `String:` **`if E1 then if E2 then S1 else S2`**

```
1: STMT -> if EXPR then STMT
2:       |  if EXPR then STMT else STMT
3:       |  s1
4:       |  s2
5: EXPR -> e1 | e2
```

Ambiguity Fixing - Exercise

Exercise: *Is this grammar ambiguous? Draw parse trees for the following* **String:** `if E1 then if E2 then S1 else S2`

```
1: STMT -> if EXPR then STMT
2:       |  if EXPR then STMT else STMT
3:       |  s1
4:       |  s2
5: EXPR -> e1 | e2
```



Ambiguity Fixing - Exercise

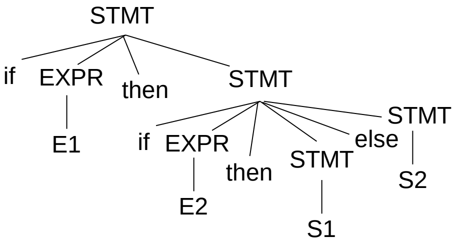
Exercise: *Is this grammar ambiguous? Draw parse trees for the following* **String: `if E1 then if E2 then S1 else S2`**

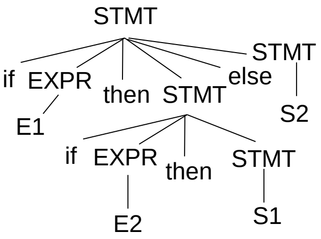
```
1: STMT -> if EXPR then STMT
2:       |  if EXPR then STMT else STMT
3:       |  s1
4:       |  s2
5: EXPR -> e1 | e2
```

Ambiguity Fixing - Exercise

Exercise: *Is this grammar ambiguous? Draw parse trees for the following* **String: if E1 then if E2 then S1 else S2**

```
1: STMT -> if EXPR then STMT
2:       |  if EXPR then STMT else STMT
3:       |  s1
4:       |  s2
5: EXPR -> e1 | e2
```

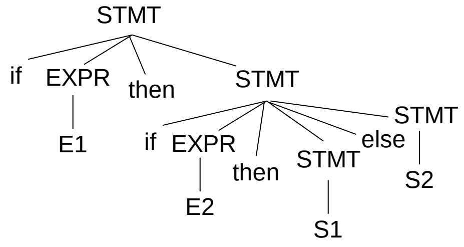




Ambiguity Fixing - Exercise

Exercise: Which `if` is the `else` associated with?

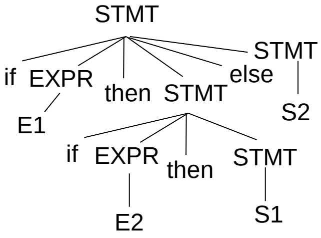
String: `if E1 then if E2 then S1 else S2`

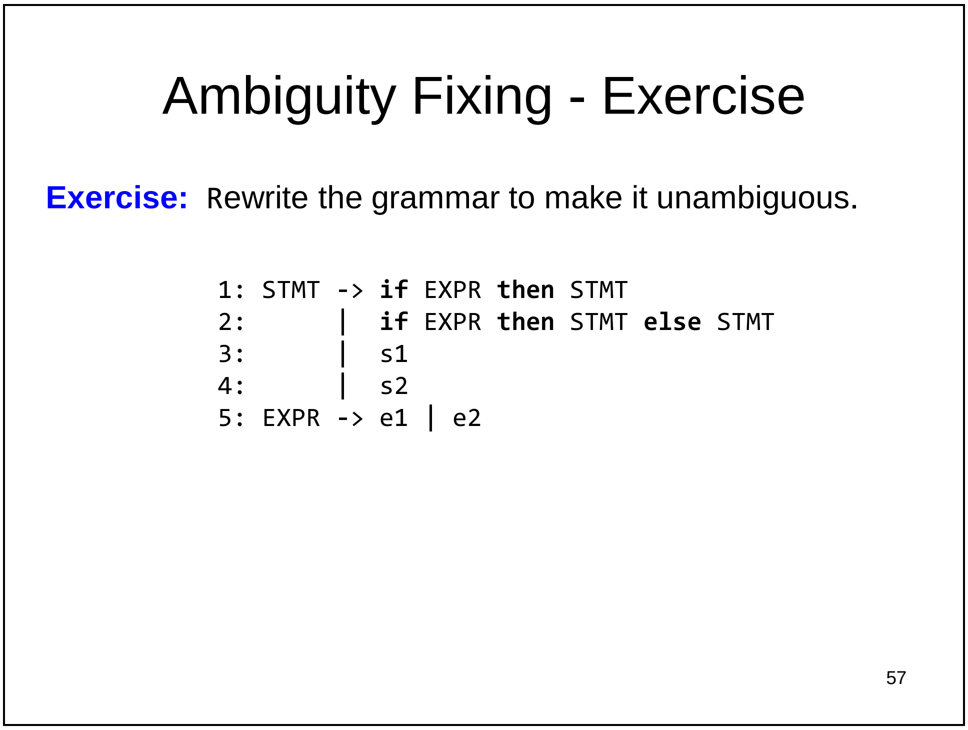


Ambiguity Fixing - Exercise

Exercise: Which `if` is the `else` associated with?

String: `if E1 then if E2 then S1 else S2`

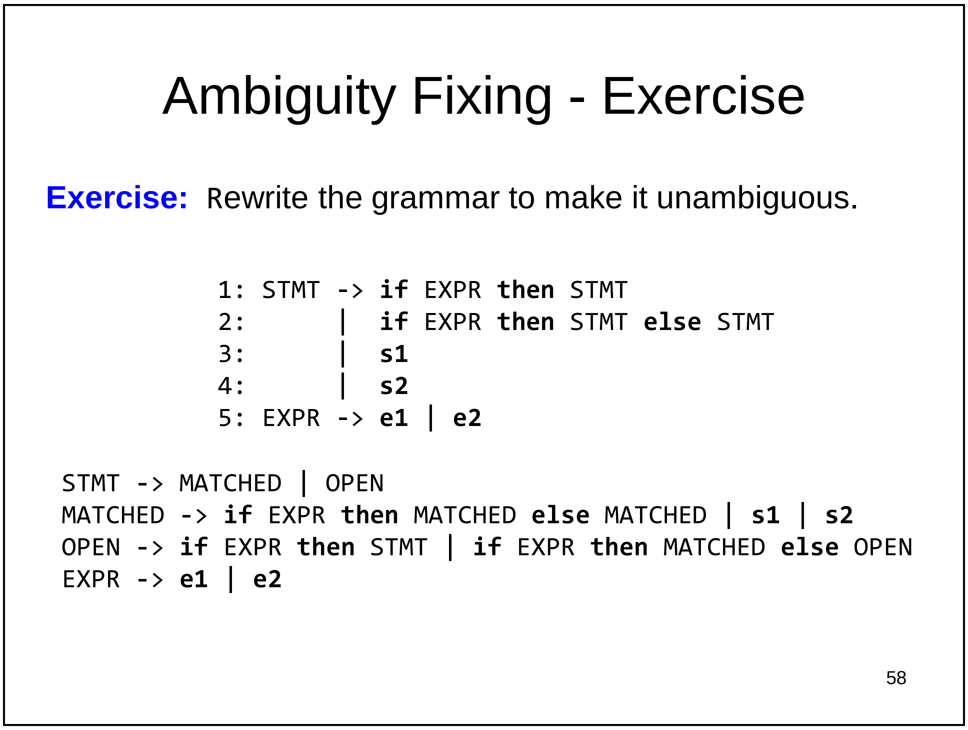




Ambiguity Fixing - Exercise

Exercise: Rewrite the grammar to make it unambiguous.

```
1: STMT -> if EXPR then STMT
2:       |  if EXPR then STMT else STMT
3:       |  s1
4:       |  s2
5: EXPR -> e1 | e2
```

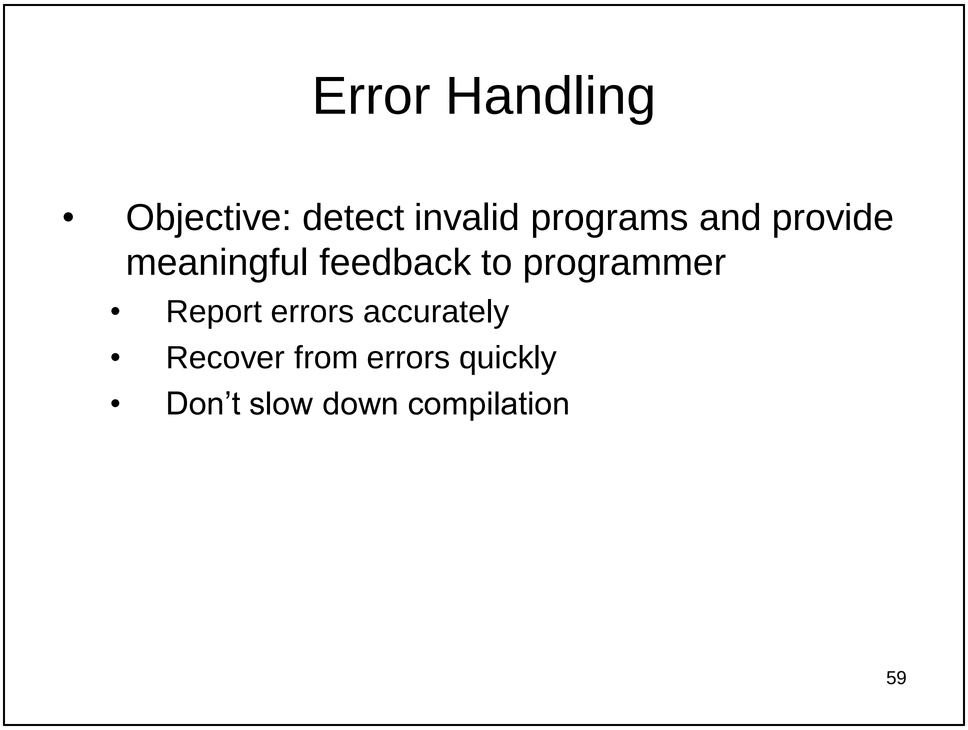


Ambiguity Fixing - Exercise

Exercise: Rewrite the grammar to make it unambiguous.

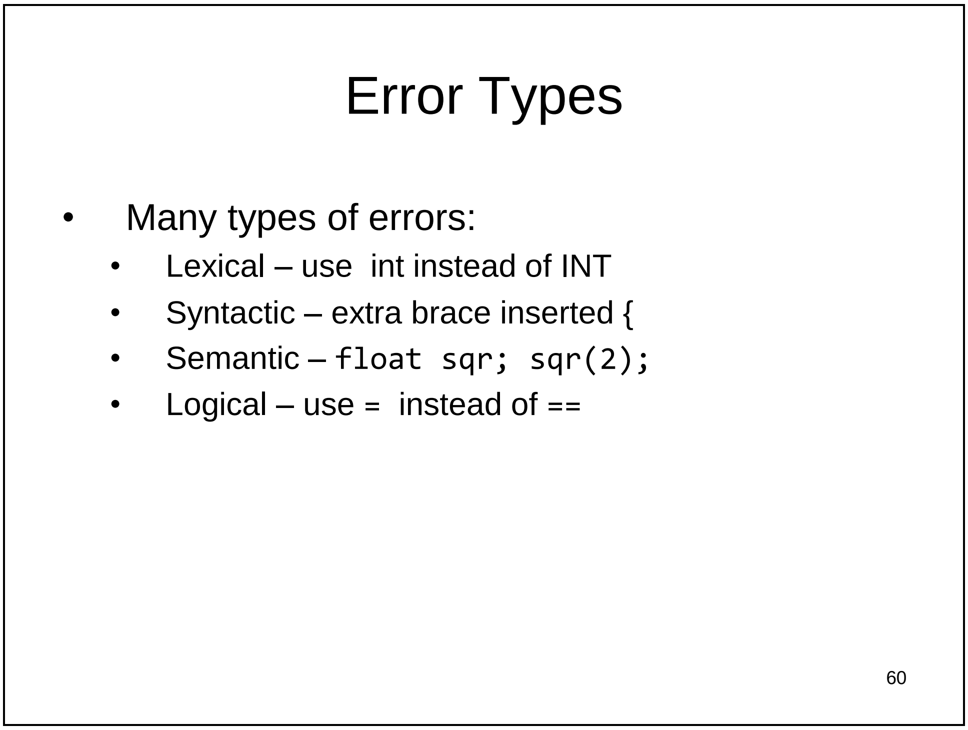
```
1: STMT -> if EXPR then STMT
2:       |  if EXPR then STMT else STMT
3:       |  s1
4:       |  s2
5: EXPR -> e1 | e2
```

```
STMT -> MATCHED | OPEN
MATCHED -> if EXPR then MATCHED else MATCHED | s1 | s2
OPEN -> if EXPR then STMT | if EXPR then MATCHED else OPEN
EXPR -> e1 | e2
```



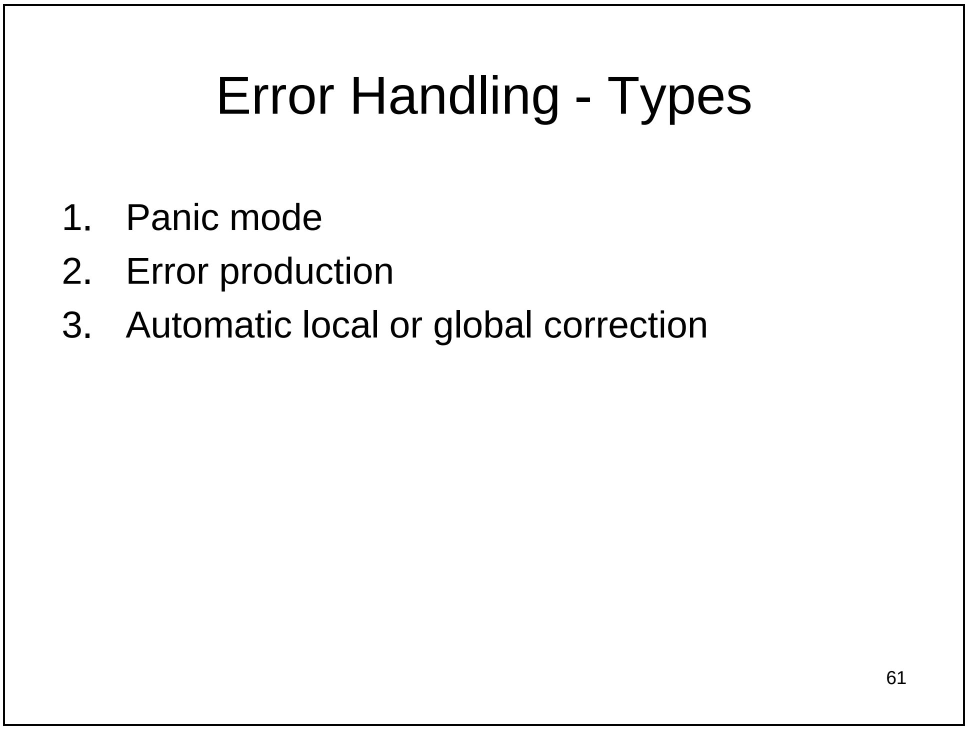
Error Handling

- Objective: detect invalid programs and provide meaningful feedback to programmer
  - Report errors accurately
  - Recover from errors quickly
  - Don’t slow down compilation



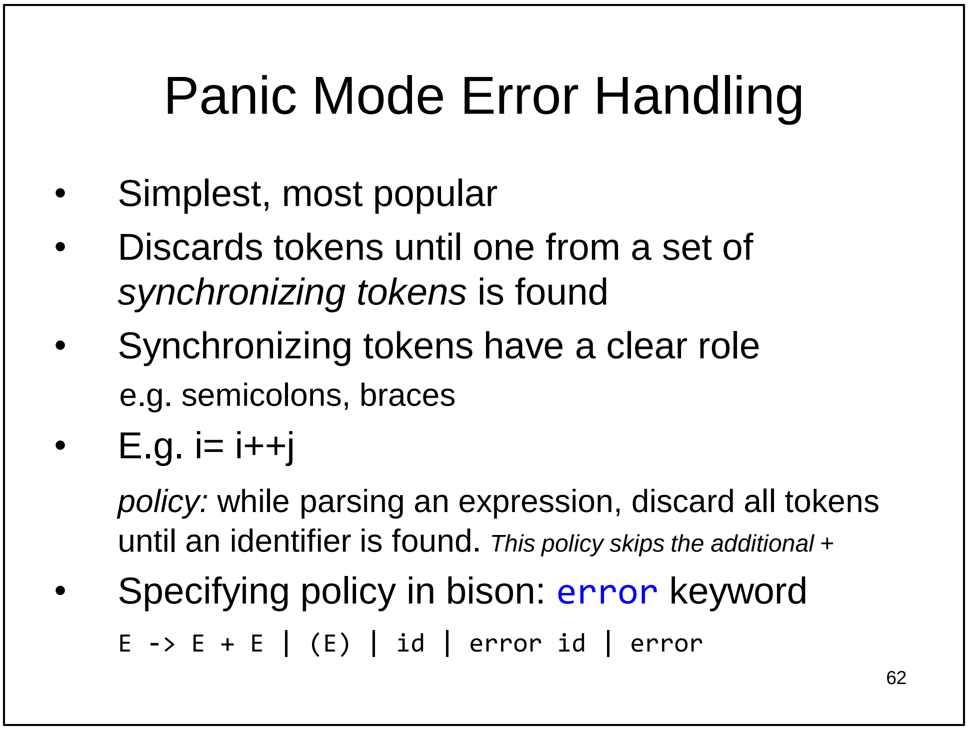
Error Types

- Many types of errors:
  - Lexical – use int instead of INT
  - Syntactic – extra brace inserted {
  - Semantic – `float sqr; sqr(2);`
  - Logical – use `=` instead of `==`



Error Handling - Types

1. Panic mode
2. Error production
3. Automatic local or global correction



Panic Mode Error Handling

- Simplest, most popular
- Discards tokens until one from a set of *synchronizing tokens* is found
- Synchronizing tokens have a clear role

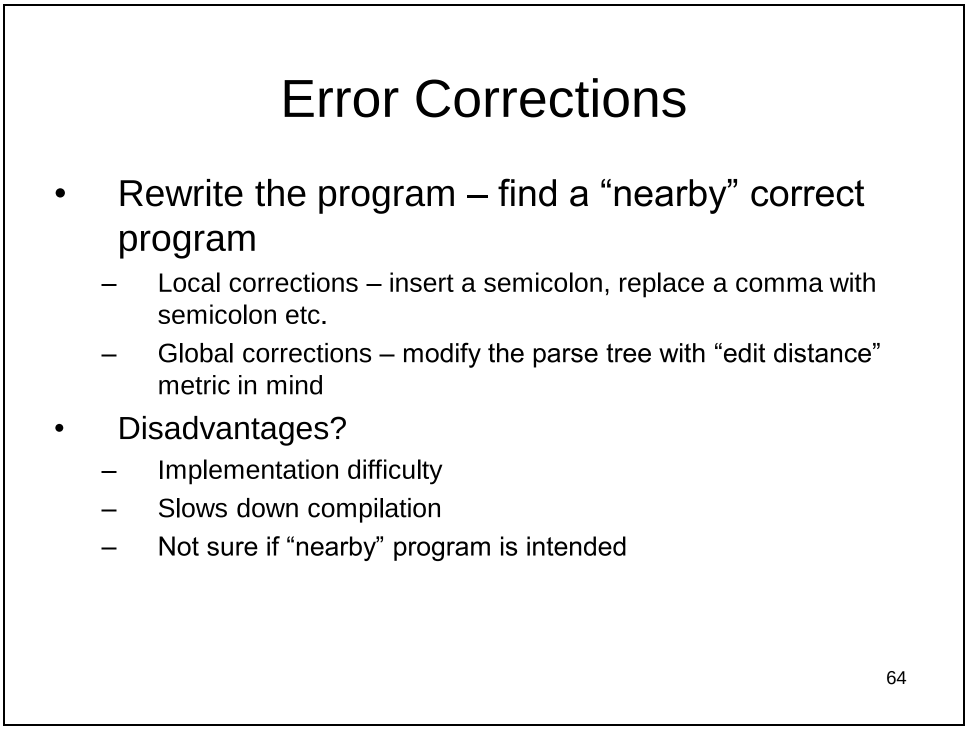
  e.g. semicolons, braces
- E.g. i= i++j

  policy: while parsing an expression, discard all tokens until an identifier is found. *This policy skips the additional* +
- Specifying policy in bison: `error` keyword

```
E -> E + E | (E) | id | error id | error
```

Error Productions

- Anticipate common errors
  - 2 x instead of 2 *
- Augment the grammar
  - E -> EE | …
- Disadvantages:
  - Complicates the grammar



Error Corrections

- Rewrite the program – find a “nearby” correct program
  - Local corrections – insert a semicolon, replace a comma with semicolon etc.
  - Global corrections – modify the parse tree with “edit distance” metric in mind
- Disadvantages?
  - Implementation difficulty
  - Slows down compilation
  - Not sure if “nearby” program is intended