CS406: Compilers

Spring 2022

Week 3: Scanners (conclusion), Parsers

Quiz_14_1 Discussion (Regular Expressions)

1. (s|p|m)(a|o)(n|g)(k|g|b|h)(r|a|i)(a|l|h)(n|u)*(ti)*

   matches “sankran”, “mankran” etc.

2. b?ho(g|l)i

   matches `hogi` etc.

3. Sankranti | Christmas | Rath Yatra | Bhai Duj | Shivaji Jayanthi

   incorrect: (Christmas / Shivaji Jayanthi / Bhai Duj / Rath Yatra)

4. lohri|pongal|Sankranti

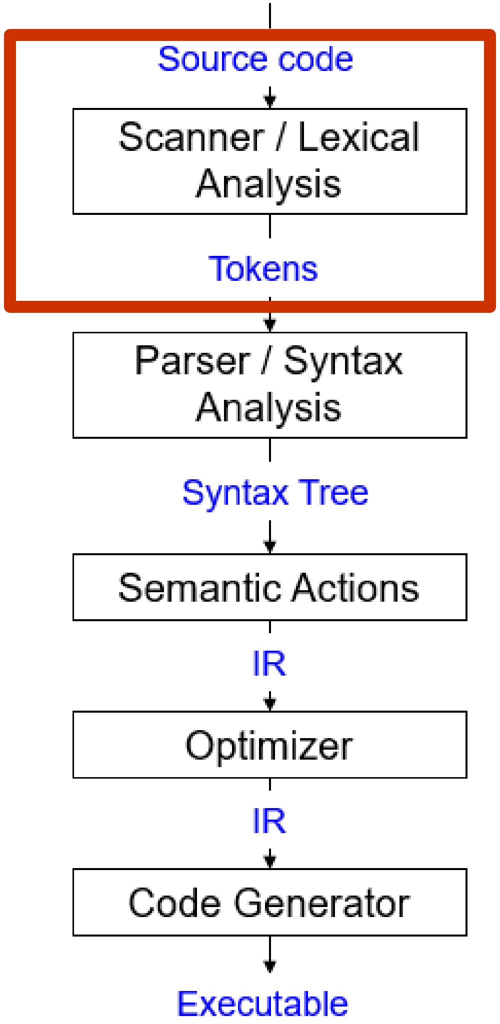
   (only 3 correct answers)

5. Pongal, sankranti,magha, bihu

   (Incorrect regular expression. Matches “Pongal,Sankranti,magha,bihu”)

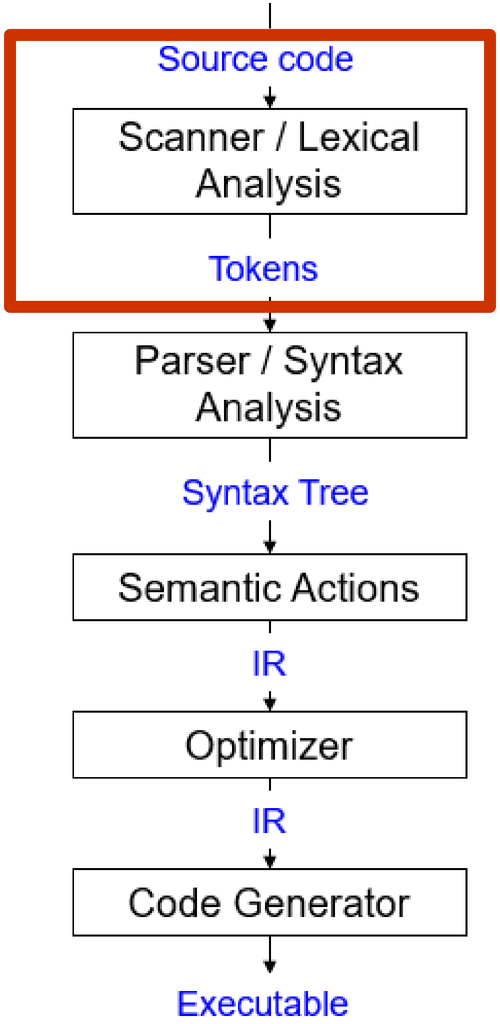
Scanners (Summary)

- Also called *Lexers* / *Lexical Analyzers*
- Input: stream of letters (program text / source code), Output: sequence / list of *tokens*
- Token: a pair <category/class, value>
  - Category defines a string *pattern*
  - Value also called *lexeme*
  - Value is a *prefix* (and hence, is a substring)
  - Value matches on of the patterns that category defines
- Scan *left-to-right* in program text, *look-ahead* to identify tokens.
  - Look-ahead buffer size determined by language design



Scanners (Summary)

- *Regular expressions* are used to formally define the patterns specified by token classes.
  - Some customization done while defining regular expressions: 1) Match the longest substring possible 2) Handle errors
- Tools such as Flex and ANTLR convert regular expressions to code. The code is your scanner implementation
  - The implementation typically converts regular expressions to *Finite Automata* (special kind of state diagram)
    - Automata are coded using efficient algorithms (E.g. Table-lookup method)
  - Efficient algorithms exist for substring matching (requiring single-pass over input program text)
    - Aho-Corasic, Knuth-Morris-Pratt (KMP)



Parsers - Overview

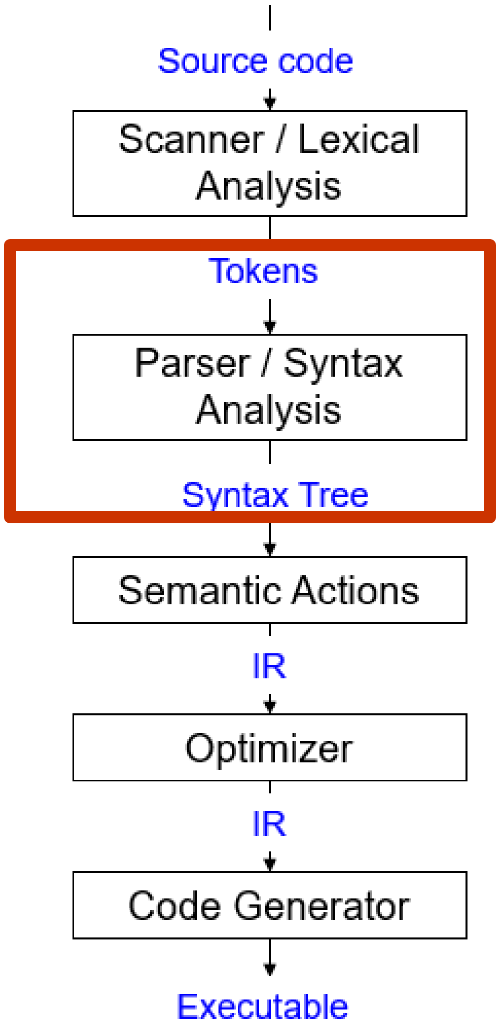
- Also called syntax analyzers
- Determine two things:
  1. Is a program syntactically valid?

     (Analogy) is an English language 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?

     (Analogy) diagramming English sentences



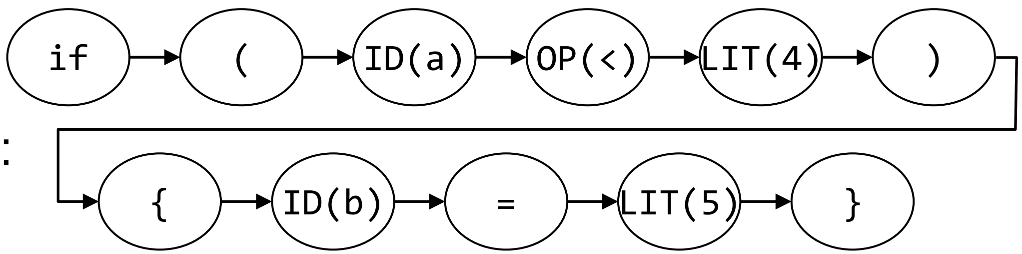
* Correponding program text:

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

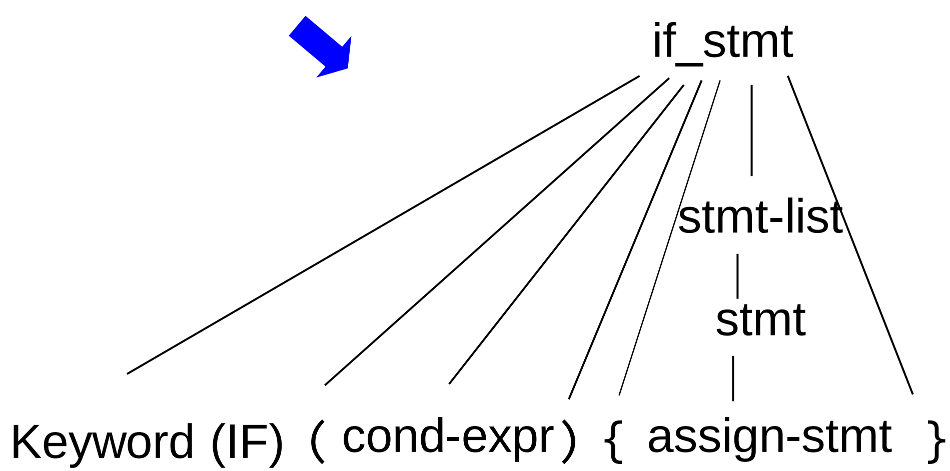
Parsers - Overview

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

Stream of tokens:



Parse tree:



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

Languages

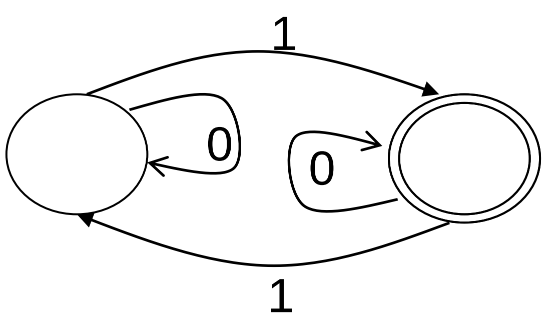
- A language is (possibly infinite) set of strings
- Regular expressions specify *regular languages.* However, regular languages are *weak formal languages* to describe the features of a practical programming language.

    What set of strings does this FA accept?

    The FA shown accepts all string with *odd number of 1s.*

    What is the regular expression for the FA?

    (0*10*)(10*10*)*



Regular expressions can describe strings specifying *parity*:

`{ mod k | k=# states in FA}`

weakness: regular expressions can't describe a string of the form: $\{ (^i\ )^i \mid i>=1\}$

Regular Languages

- Regular expressions can’t describe a string of the form:

$$\{ (^i)^i \mid i>=1\}$$

E.g. Parenthesized expressions

```
((2+3)*5)

((( int x; )))
```

Programming language syntax is i.e. recursive

Nested structures:

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

Context Free Grammar (CFG)

- Natural notation for describing <u>recursive structure</u> 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* ($X \rightarrow Y_1..Y_N$) (aka. rules)

$P: X \longrightarrow Y_1Y_2Y_3..Y_N \qquad X \in N, \qquad Y_i \in N \cup T \cup \epsilon/\lambda$

Context Free Grammar (CFG)

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

  E.g. $G = (\{a,b\}, \{S, A, B\}, S, \{S \rightarrow AB,\ A \rightarrow Aa,$
  $A \rightarrow a,\ B \rightarrow Bb,\ B \rightarrow b\})$

- Implicit meanings
  - <u>First rule</u> listed in the set of productions contains <u>start symbol</u> (on the left-hand side)
  - In the set of productions, <u>you can replace the symbol X</u> (appearing on the right-hand side only) with the <u>string of symbols</u> 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
   i. AB replaced with aB
   ii. aB replaced with ab

Summary: we move from S to a string of terminals through a series of <u>transformations:</u>

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

Shorthand notation: $\alpha_0 \overset{*}{\rightarrow} \alpha_n$

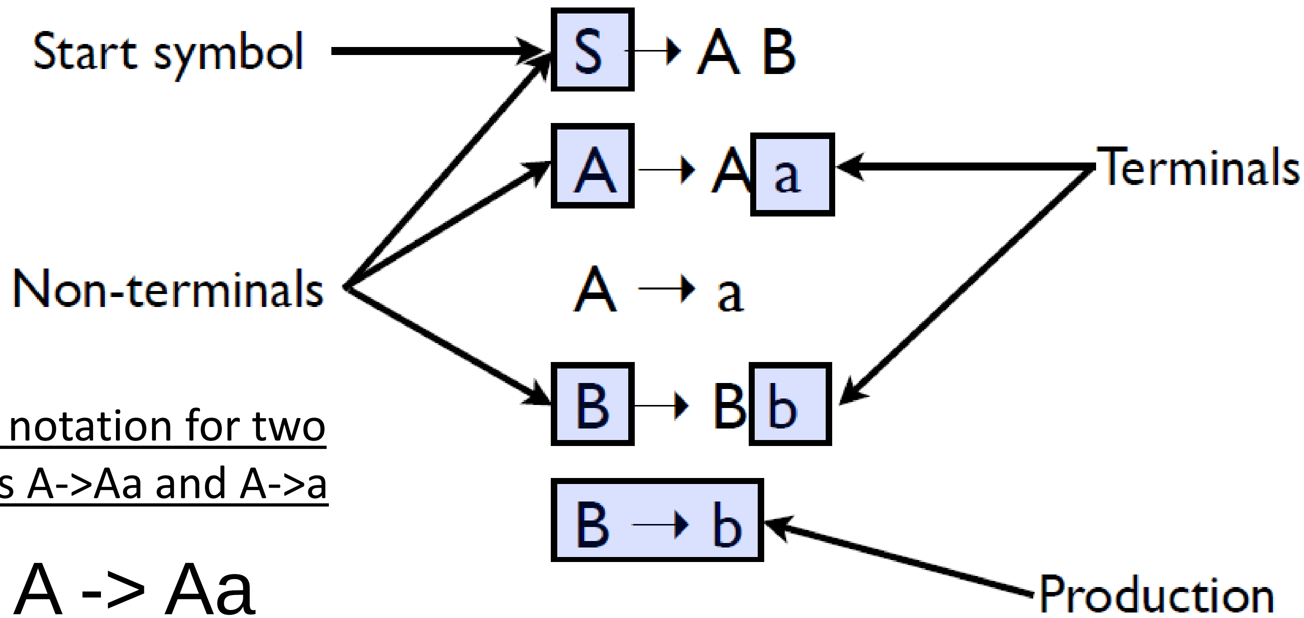
Detour: Context-Sensitive Grammar

- 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 }
```

Simple grammar (Summary)



Alternative notation for two productions A->Aa and A->a

A -> Aa
| a

Backus Naur Form (BNF)

Slide courtesy: Milind Kulkarni

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

Language of the Grammar

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

String Derivations: Does a string belong to the Language?

- How do we apply the grammar rules to determine the acceptability of a string? (i.e. the string belongs to the language, *L(G)*, specified by the CFG *G*)
  - Begin with S
  - Replace S
  - Repeat till string contains terminals only. Why terminals only?

L(G) must contain strings of terminals only

- Notation:
  - We will use Greek letters to denote strings containing non-terminals and terminals
- Derivations: sequence of rules applied to produce the string of terminals

Generating strings (Example)

$S \rightarrow A\ B$

$A \rightarrow A\ a$

$A \rightarrow a$

$B \rightarrow B\ b$

$B \rightarrow b$

- 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:

$$S \Rightarrow A\ B \Rightarrow A\ a\ B \Rightarrow a\ a\ B \Rightarrow a\ a\ B\ b \Rightarrow$$
$$a\ a\ B\ b\ b \Rightarrow a\ a\ b\ b\ b$$

CFG and Parsers

- 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

Exercise

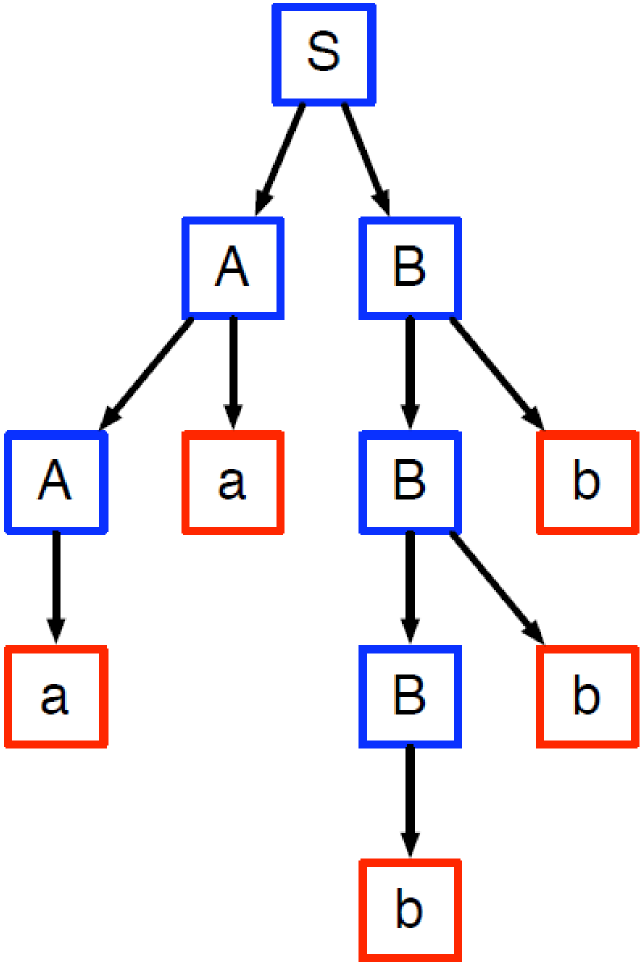
Which of the below strings are accepted by the grammar:

```
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

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



Derivations and Parse Trees

- Recall: Derivation is a sequence of rules applied to produce a string
  - $S \to \alpha_0 \to \alpha_1 \to \alpha_2 \to \ldots \to \alpha_n$
- A derivation defines a parse tree
  - Parse tree is an alternative way to gather information on how the string was derived
  - A parse tree may have many derivations (think: different permutations of α)

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

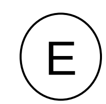
- Consider the grammar with the following rules:

```
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:

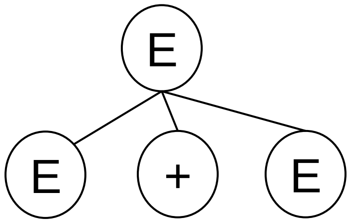
```
1: E -> E + E
2:      | E * E
3:      | id
```

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

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

E
E+E

Parse Tree



Derivations and Parse Trees

- Consider the grammar with the following rules:

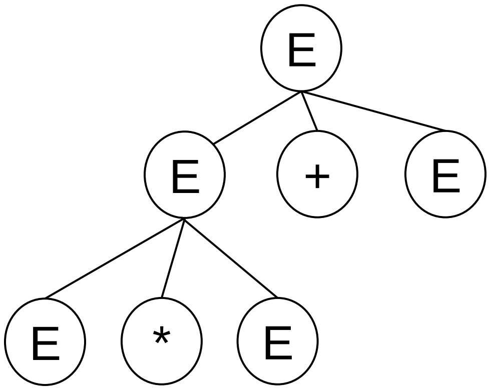
```
1: E -> E + E
2:      | E * E
3:      | id
```

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

Apply 2: Replace E with E * E

E
E+E
E*E+E

Parse Tree



Derivations and Parse Trees

- Consider the grammar with the following rules:

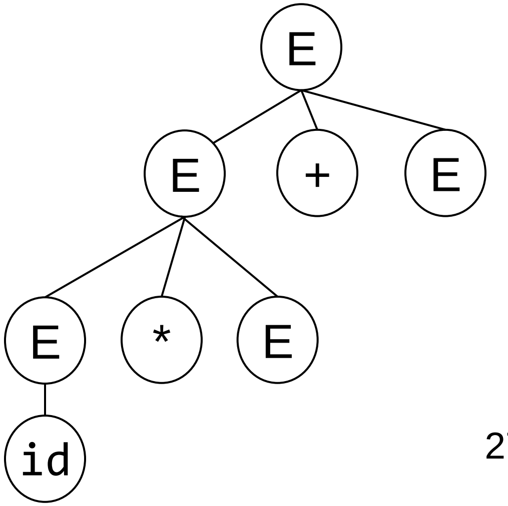
```
1: E -> E + E
2:      | E * E
3:      | id
```

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

Apply 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:

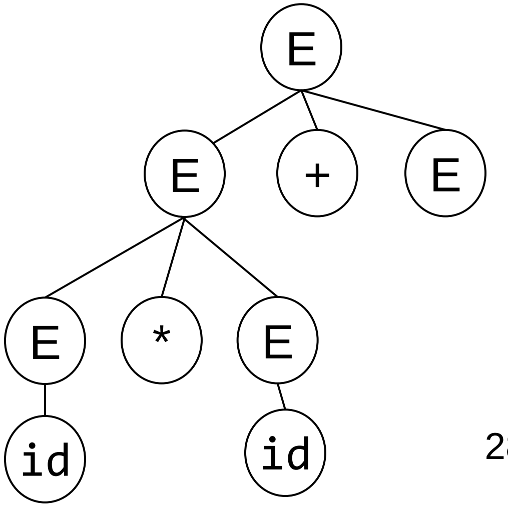
```
1: E -> E + E
2:      | E * E
3:      | id
```

- 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:

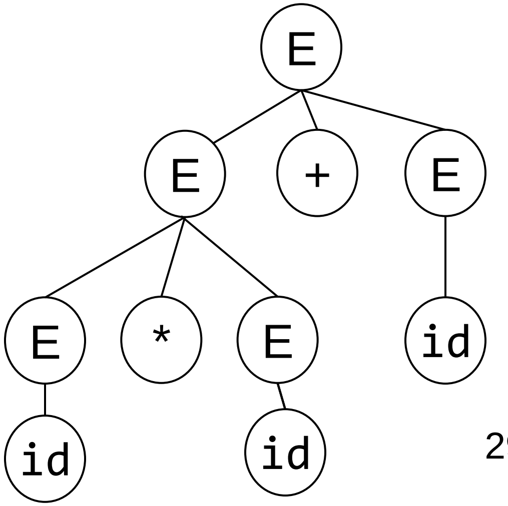
```
1: E -> E + E
2:      | E * E
3:      | id
```

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

Apply 3: Replace E with id

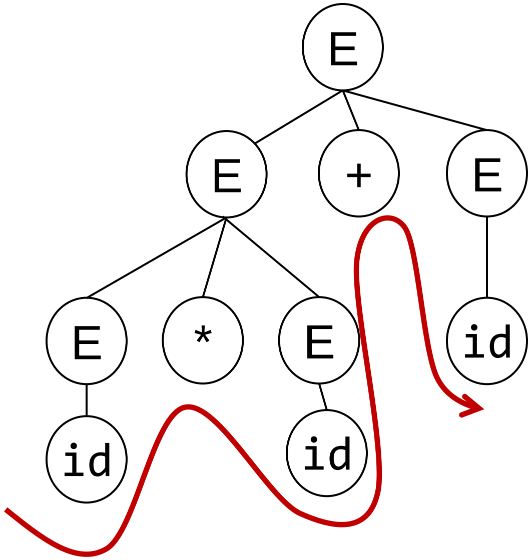
```
E
E+E
E*E+E
id*E+E
id*id+E
id*id+id
```

Parse Tree



Derivations and Parse Trees

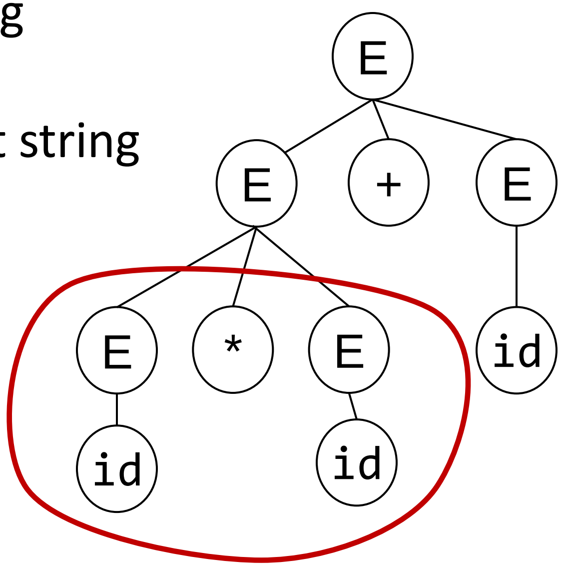
- Note in previous slides:
  - Replacement done on left-most non-terminal in the string - **called left-most derivation**
  - Terminals at leaves and non-terminal as interior nodes
  - `Inorder` traversal of leaves produces input string `id*id+id`



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 of leaves produces input string `id*id+id`
  - Parse tree shows <u>association of operations</u>. Input string doesn't
    - `*` associated with identifiers in the subtree

`(id * id)+id`



Derivations and Parse Trees

- Consider the same grammar (having 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

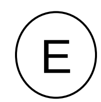
- Consider the grammar with the following rules:

```
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:

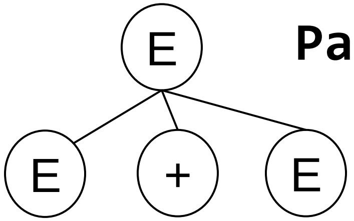
```
1: E -> E + E
2:      | E * E
3:      | id
```

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

Apply 2: **Replace E with E+E**

E

E+E



Parse Tree

Derivations and Parse Trees

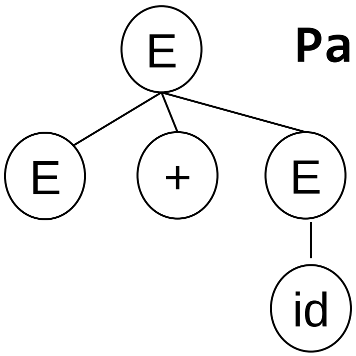
- Consider the grammar with the following rules:

```
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

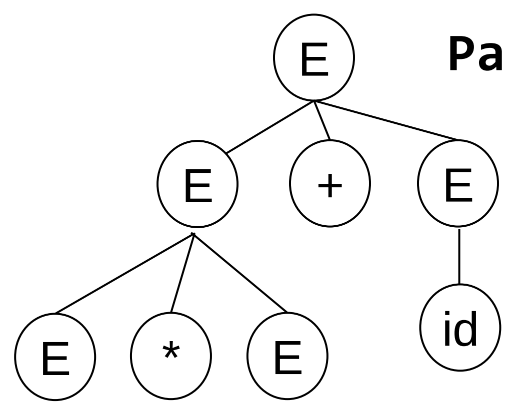
- Consider the grammar with the following rules:

```
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



Parse Tree

Derivations and Parse Trees

- Consider the grammar with the following rules:

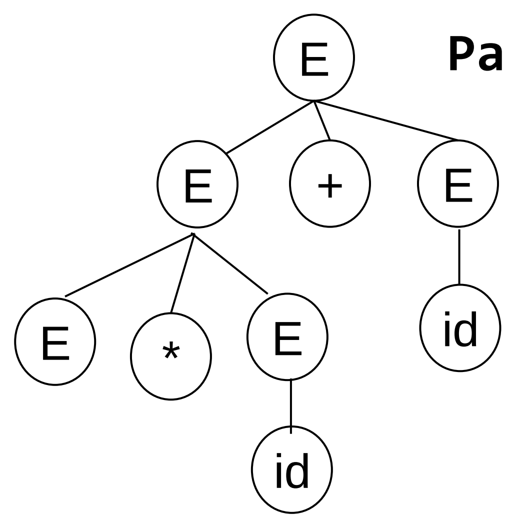
```
1: E -> E + E
2:      | E * E
3:      | id
```

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

Apply 3: **Replace E with id**

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

Parse Tree



Derivations and Parse Trees

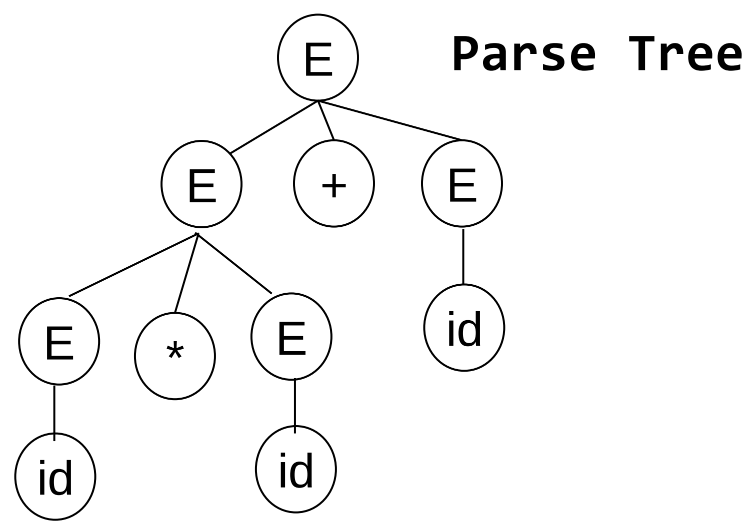
- Consider the grammar with the following rules:

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

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

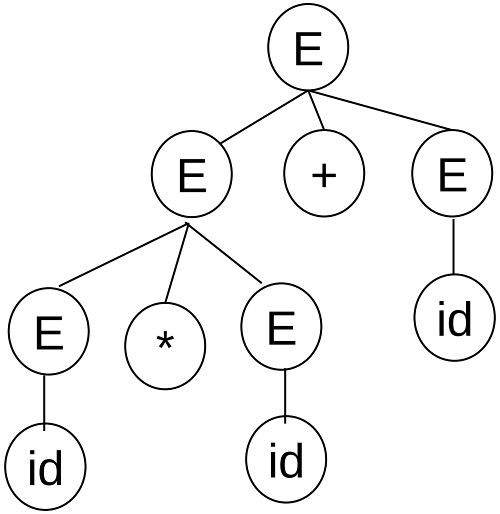
Apply 3: **Replace E with id**

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



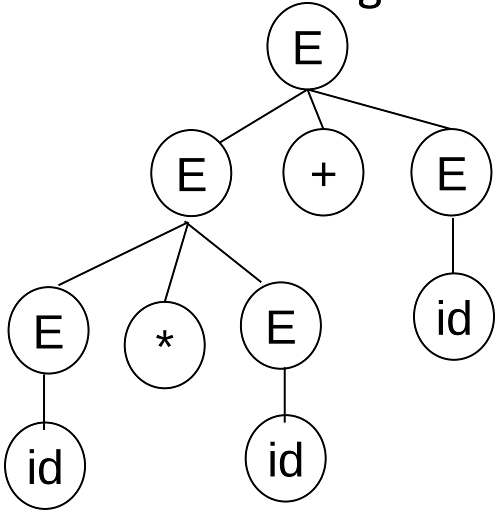
Derivations and Parse Trees

- 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

E

Parse Tree

Derivations and Parse Trees

- Consider the grammar with the following rules:

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

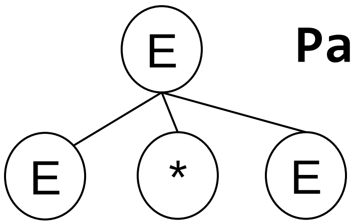
- Produce derivations for the string: **`id*id+id`**

Earlier it was replace E with E+E

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

E
E*E

Parse Tree



Derivations and Parse Trees

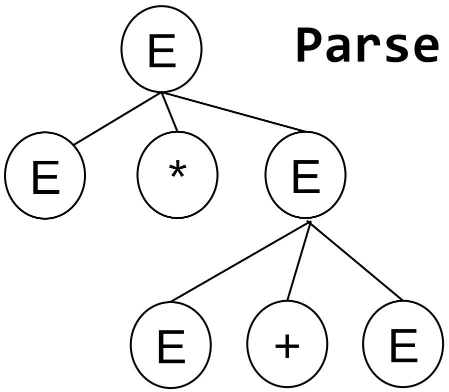
- Consider the grammar with the following rules:

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

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

Apply 1: Replace E with E+E

E
E*E
E*E+E



Parse Tree

Derivations and Parse Trees

- Consider the grammar with the following rules:

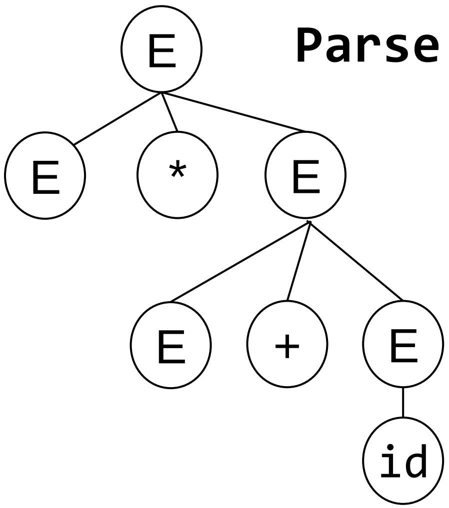
```
1: E -> E + E
2:    | E * E
3:    | id
```

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

Apply 3: Replace E with id

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

Parse Tree



Derivations and Parse Trees

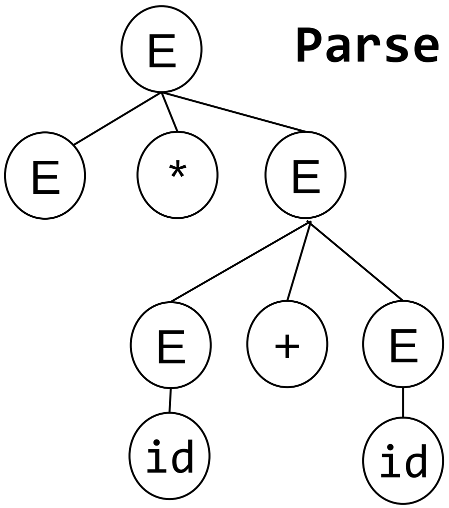
- Consider the grammar with the following rules:

```
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:

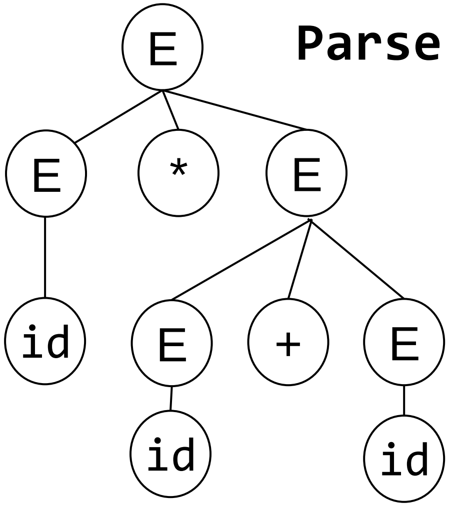
```
1: E -> E + E
2:      | E * E
3:      | id
```

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

Apply 3: **Replace E with id**

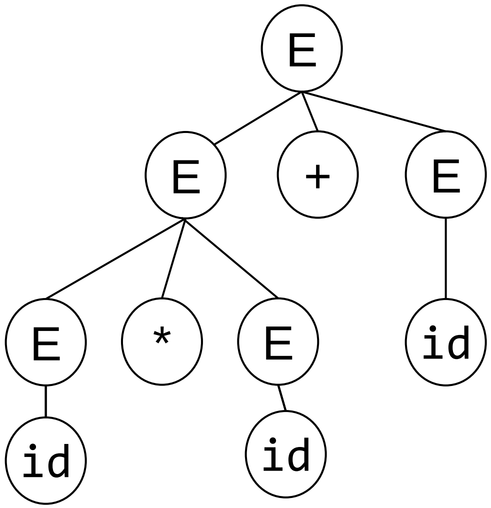
E
E*E
E*E+E
E*E+id
E*id+id
id*id+id

Parse Tree

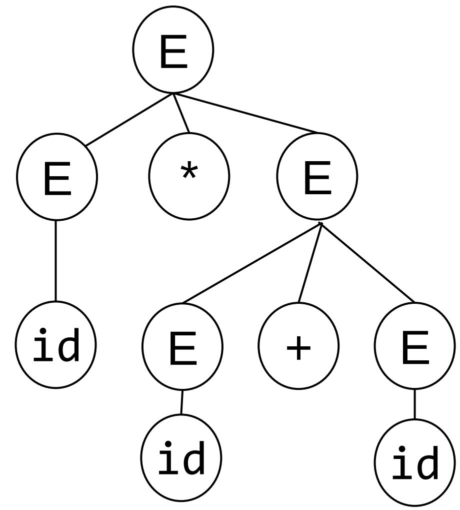


Derivations and Parse Trees

- Input string: `id*id+id`



earlier



now

- `Inorder` traversal of leaves in both trees produces the same input string

Ambiguous Grammar

- Grammar that produces more than one parse tree for <u>some</u> 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