CS345H: Programming Languages

Lecture 5: Introduction to Parsing

Thomas Dillig

Outline

- Limitations of Regular Languages

Outline

- Limitations of Regular Languages
- Parser Overview

Outline

- Limitations of Regular Languages
- Parser Overview
- Context-free Grammars (CFGs)

Outline

- Limitations of Regular Languages
- Parser Overview
- Context-free Grammars (CFGs)
- Derivations

Outline

- Limitations of Regular Languages
- Parser Overview
- Context-free Grammars (CFGs)
- Derivations
- Ambiguity

Regular Languages

- Last time, we saw that regular languages are very useful for partitioning input into tokens

Regular Languages

- Last time, we saw that regular languages are very useful for partitioning input into tokens
- But regular languages are not expressive enough to turn a stream of tokens into structure

Regular Languages

- Last time, we saw that regular languages are very useful for partitioning input into tokens
- But regular languages are not expressive enough to turn a stream of tokens into structure
- For this, we need a more expressive formal language

Beyond Regular Languages

- Many languages are not regular

Beyond Regular Languages

- Many languages are not regular
- Classic Example: Strings of balanced parenthesis:

$$\{(^i)^j \mid i \geq 0\}$$

Beyond Regular Languages

- Many languages are not regular
- Classic Example: Strings of balanced parenthesis:

$$\{(^i)^j \mid i \geq 0\}$$

- Question: Why is there no automata that can recognize this language?

What Can Regular Languages Express?

- Languages requiring counting modulo a fixed integer

What Can Regular Languages Express?

- Languages requiring counting modulo a fixed integer
- Intuition: A finite automaton that runs long enough must repeat states

What Can Regular Languages Express?

- Languages requiring counting modulo a fixed integer
- Intuition: A finite automaton that runs long enough must repeat states
- Finite automaton cannot remember the number of times it has visited a particular state

Side Note: Comments in L

- Recall: Comments in L start with `(*`, end with `*)` and can be nested

Side Note: Comments in L

- Recall: Comments in L start with `(*`, end with `*)` and can be nested
- Also Recall: Comments are removed during lexing

Side Note: Comments in L

- Recall: Comments in L start with `(*`, end with `*)` and can be nested
- Also Recall: Comments are removed during lexing
- Question: Are comments in L a regular language?

The Functionality of the Parser

- Input: sequence of tokens from the lexer

The Functionality of the Parser

- Input: sequence of tokens from the lexer
- Output: parse tree of the program

Example

- Consider the following L expression:
  `if x<>y then 1 else 2`

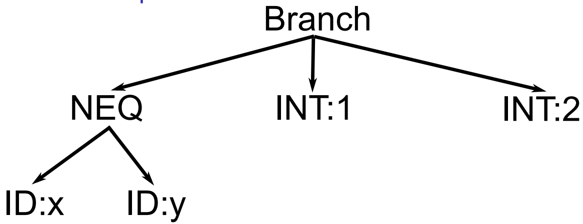
Example

- Consider the following L expression:
  `if x<>y then 1 else 2`

- Parse Input: `TOKEN_IF TOKEN_ID("x") TOKEN_NEQ TOKEN_ID("y") TOKEN_THEN TOKEN_INT(1) TOKEN_ELSE TOKEN_INT(2)`

Example

- Consider the following L expression:
  `if x<>y then 1 else 2`
- Parse Input: `TOKEN_IF TOKEN_ID("x") TOKEN_NEQ TOKEN_ID("y") TOKEN_THEN TOKEN_INT(1) TOKEN_ELSE TOKEN_INT(2)`
- Parser Output:



Parsing vs. Lexing

Phase	Input	Output
Lexer		
Parser		

Parsing vs. Lexing

Phase	Input	Output
Lexer	String of characters	
Parser		

Parsing vs. Lexing

Phase	Input	Output
Lexer	String of characters	String of tokens
Parser		

Parsing vs. Lexing

Phase	Input	Output
Lexer	String of characters	String of tokens
Parser	String of tokens	

Parsing vs. Lexing

Phase	Input	Output
Lexer	String of characters	String of tokens
Parser	String of tokens	Parse tree

The Role of the Parser

- Not all strings of tokens are programs . . .

The Role of the Parser

- Not all strings of tokens are programs . . .
- Parser must distinguish between valid and invalid strings of tokens

The Role of the Parser

- Not all strings of tokens are programs ...
- Parser must distinguish between valid and invalid strings of tokens
- We need:

The Role of the Parser

- Not all strings of tokens are programs ...
- Parser must distinguish between valid and invalid strings of tokens
- We need:
  - A language for describing valid strings of tokens

The Role of the Parser

- Not all strings of tokens are programs . . .
- Parser must distinguish between valid and invalid strings of tokens
- We need:
  - A language for describing valid strings of tokens
  - A method for recognizing if a string of tokens is in this language or not

Context-free Grammars (CFGs)

- Programming language constructs have recursive structure

Context-free Grammars (CFGs)

- Programming language constructs have recursive structure
- Example: An L expression is `expression + expression`, `if expression then expression else expression`, ...

Context-free Grammars (CFGs)

- Programming language constructs have recursive structure
- Example: An L expression is `expression + expression`, `if expression then expression else expression`, ...
- Context free grammars are a natural notation for this recursive structure

CFGs in more detail

- A CFG consists of:

CFGs in more detail

- A CFG consists of:
  - A set of terminals T

CFGs in more detail

- A CFG consists of:
  - A set of terminals T
  - A set of non-terminals N

CFGs in more detail

- A CFG consists of:
  - A set of terminals T
  - A set of non-terminals N
  - A start symbol S (non-terminal)

CFGs in more detail

- A CFG consists of:
  - A set of terminals T
  - A set of non-terminals N
  - A start symbol S (non-terminal)
  - A set of productions

$$X \rightarrow Y_1 Y_2 \ldots Y_n$$

where $X \in N$ and $Y_i \in (T \cup N \cup \{\varepsilon\})$

Notational Conventions in this Class

- Non-terminals are always written upper-case

Notational Conventions in this Class

- Non-terminals are always written upper-case
- Terminals are written lower-case

Notational Conventions in this Class

- Non-terminals are always written upper-case
- Terminals are written lower-case
- The start symbol is the left-hand side of the first production

CFG Examples

- A fragment of L

CFG Examples

- A fragment of L

```
EXPR → if EXPR then EXPR else EXPR
        |  EXPR + EXPR
        |  id
```

CFG Examples continued

- Simple arithmetic expressions:

CFG Examples continued

- Simple arithmetic expressions:

```
EXPR → E * E
        | E + E
        | (E)
        | id
```

The Language of a CFG

- Recall production rules: $X \to Y_1 \dots Y_n$

The Language of a CFG

- Recall production rules: $X \rightarrow Y_1 \ldots Y_n$
- Means that X can be replaced by $Y_1 \ldots Y_N$

The Language of a CFG

- Recall production rules: $X \rightarrow Y_1 \dots Y_n$
- Means that X can be replaced by $Y_1 \dots Y_N$
- More specifically:

The Language of a CFG

- Recall production rules: $X \rightarrow Y_1 \ldots Y_n$
- Means that X can be replaced by $Y_1 \ldots Y_N$
- More specifically:
  1. Begin with string consisting of the start symbol "S"

The Language of a CFG

- Recall production rules: $X \to Y_1 \ldots Y_n$
- Means that X can be replaced by $Y_1 \ldots Y_N$
- More specifically:
  1. Begin with string consisting of the start symbol "S"
  2. Replace any non-terminal X in string with the right-hand side of some production

$$X \to Y_1 \ldots Y_n$$

The Language of a CFG

- Recall production rules: $X \to Y_1 \ldots Y_n$
- Means that X can be replaced by $Y_1 \ldots Y_N$
- More specifically:
  1. Begin with string consisting of the start symbol "S"
  2. Replace any non-terminal X in string with the right-hand side of some production

     $$X \to Y_1 \ldots Y_n$$

  3. Repeat (2) until there are no non-terminals in the string

The Language of a CFG continued

- More formally, write

$$X_1 \dots X_i \dots X_n \rightarrow X_1 \dots X_{i-1} \, Y_1 \dots Y_m X_{i+1} \dots X_n$$

if there is a production

$$X_i \rightarrow Y_1 \dots Y_m$$

The Language of a CFG continued

- More formally, write

  $$X_1 \ldots X_i \ldots X_n \rightarrow X_1 \ldots X_{i-1} Y_1 \ldots Y_m X_{i+1} \ldots X_n$$

  if there is a production

  $$X_i \rightarrow Y_1 \ldots Y_m$$

- Abbreviation: Write $X_1 \ldots X_n \rightarrow^* Y_1 \ldots Y_m$ if $X_1 \ldots X_n \rightarrow \ldots \rightarrow Y_1 \ldots Y_m$ in 0 or more steps

The Language of a CFG continued

- Now, let G be a context-free grammar with start symbol S. Then the language of G is:

$$\{a_1 \ldots a_n | S \rightarrow^* a_1 \ldots a_n \text{ and every } a_i \text{ is a terminal}\}$$

Terminals

- Terminals are called "terminals" because there are no rules for replacing them

Terminals

- Terminals are called "terminals" because there are no rules for replacing them
- Once generated, terminals are permanent

Terminals

- Terminals are called ”terminals” because there are no rules for replacing them
- Once generated, terminals are permanent
- Question: What should terminals be when parsing a programming language?

Terminals

- Terminals are called "terminals" because there are no rules for replacing them
- Once generated, terminals are permanent
- Question: What should terminals be when parsing a programming language?
- Answer: Tokens

Examples

- $L(G)$ is the language of CFG G

Examples

- $L(G)$ is the language of CFG G
- Strings of balanced parentheses:

$$\{(^i)^j | i \geq 0\}$$

Examples

- $L(G)$ is the language of CFG G
- Strings of balanced parentheses:

$$\{(^i)^j | i \geq 0\}$$

- CFG:

$$S \rightarrow (S)$$
$$S \rightarrow \varepsilon$$

Examples

- $L(G)$ is the language of CFG G
- Strings of balanced parentheses:

$$\{(^i)^j | i \geq 0\}$$

- CFG:

$$S \rightarrow (S)$$
$$S \rightarrow \varepsilon$$

or equivalently

$$S \rightarrow (S) \mid \varepsilon$$

Examples

- Recall the earlier fragment of L:

Examples

- Recall the earlier fragment of L:

```
EXPR → if EXPR then EXPR else EXPR
     |   EXPR + EXPR
     |   id
```

Examples

- Recall the earlier fragment of L:

```
EXPR → if EXPR then EXPR else EXPR
     |  EXPR + EXPR
     |  id
```

- Some strings in this language:

Examples

- Recall the earlier fragment of L:

```
EXPR → if EXPR then EXPR else EXPR
        |  EXPR + EXPR
        |  id
```

- Some strings in this language:

- ```
  ID
  IF ID THEN ID ELSE ID
  ID + ID
  IF ID THEN ID+ID ELSE ID
  IF IF ID THEN ID ELSE IF THEN ID ELSE ID
  ...
  ```
  ```

Examples

- Recall simple arithmetic expressions:

Examples

- Recall simple arithmetic expressions:

```
EXPR → E * E
     |  E + E
     |  (E)
     |  id
```

Examples

- Recall simple arithmetic expressions:

```
EXPR → E * E
     |  E + E
     |  (E)
     |  id
```

- Some strings in this language:

Examples

- Recall simple arithmetic expressions:

```
EXPR → E * E
         |  E + E
         |  (E)
         |  id
```

- Some strings in this language:

- `id`
  `(id)`
  `(id)*id`
  `id+id`
  `id*id`
  `id*(id) ...`

Where are we?

- The idea of a CFG is a big step towards parsing tokens.

Where are we?

- The idea of a CFG is a big step towards parsing tokens.
- But we don't just want to know if a string of tokens is in a language, we also need parse tree of input tokens

Where are we?

- The idea of a CFG is a big step towards parsing tokens.
- But we don't just want to know if a string of tokens is in a language, we also need parse tree of input tokens
- Must also handle errors gracefully

Where are we?

- The idea of a CFG is a big step towards parsing tokens.
- But we don't just want to know if a string of tokens is in a language, we also need parse tree of input tokens
- Must also handle errors gracefully
- Need an implementation of CFGs (e.g., bison)

From Derivations to Parse Trees

- A derivation is a sequence of productions

$$S \to \ldots \to \ldots \to \ldots$$

From Derivations to Parse Trees

- A derivation is a sequence of productions

$$S \to \ldots \to \ldots \to \ldots$$

- A derivation can be drawn as a tree

From Derivations to Parse Trees

- A derivation is a sequence of productions

$$S \to \ldots \to \ldots \to \ldots$$

- A derivation can be drawn as a tree
  - Start symbol is the tree's root

From Derivations to Parse Trees

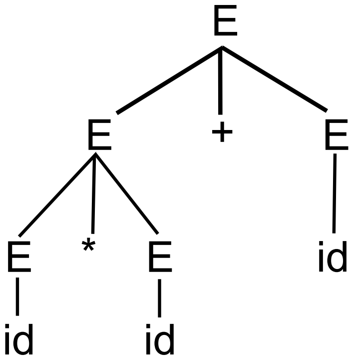
- A derivation is a sequence of productions

$$S \to \ldots \to \ldots \to \ldots$$

- A derivation can be drawn as a tree
  - Start symbol is the tree's root
  - For a production $X \to Y_1 \ldots Y_n$ add children $Y_1 \ldots Y_n$ to node X

Derivation Example

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



Derivation in Detail

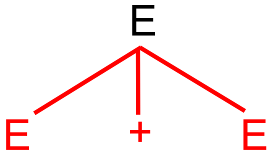
E

E

Derivation in Detail

E

$\rightarrow$ E+E

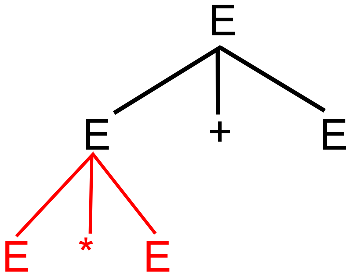


Derivation in Detail

E

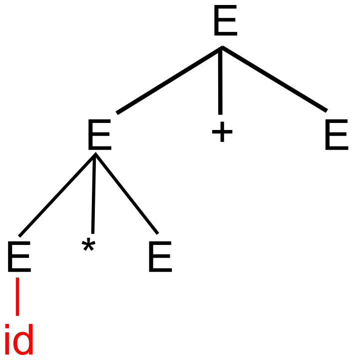
$\rightarrow$ E+E

$\rightarrow$ E*E+E



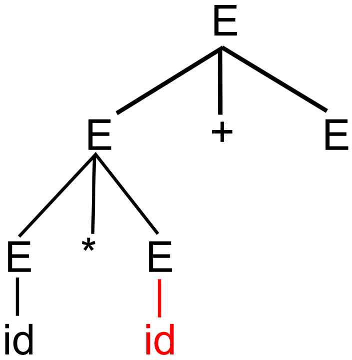
Derivation in Detail

$$\begin{array}{ll} & \texttt{E} \\ \rightarrow & \texttt{E+E} \\ \rightarrow & \texttt{E*E+E} \\ \rightarrow & \texttt{id*E+E} \end{array}$$



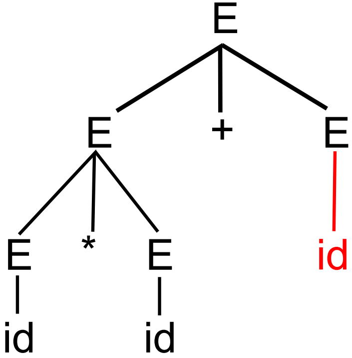
Derivation in Detail

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



Derivation in Detail

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



Notes on Derivations

- A parse tree has terminals at the leaves and non-terminals at the interior nodes

Notes on Derivations

- A parse tree has terminals at the leaves and non-terminals at the interior nodes
- An in-order traversal of the leaves is the original input

Notes on Derivations

- A parse tree has terminals at the leaves and non-terminals at the interior nodes
- An in-order traversal of the leaves is the original input
- The parse tree shows the associativity of operations, the input token string does not

Notes on Derivations

- A parse tree has terminals at the leaves and non-terminals at the interior nodes
- An in-order traversal of the leaves is the original input
- The parse tree shows the associativity of operations, the input token string does not
- Example: The parse tree from the last slide encodes that times has higher precedence than plus

Left-most and Right-most Derivations

- The example we looked at is a left-most derivation

Left-most and Right-most Derivations

- The example we looked at is a left-most derivation
- This means: At each step, we replace the left-most non-terminal

Left-most and Right-most Derivations

- The example we looked at is a left-most derivation
- This means: At each step, we replace the left-most non-terminal
- There is also an equivalent notion of right-most derivation

Right-most Derivation in Detail

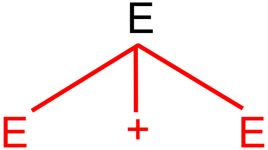
E

`E`

Right-most Derivation in Detail

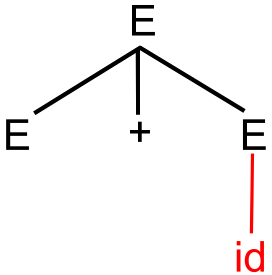
E

$\rightarrow$ E+E



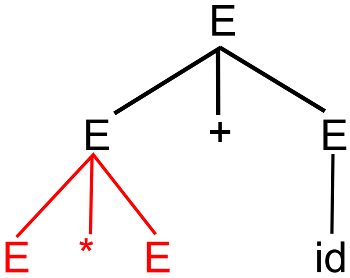
Right-most Derivation in Detail

E
$\rightarrow$ E+E
$\rightarrow$ E+id



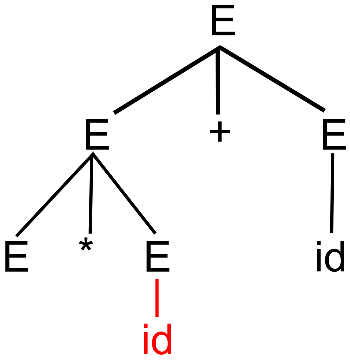
Right-most Derivation in Detail

$$\begin{aligned} & \texttt{E} \\ \rightarrow\ & \texttt{E+E} \\ \rightarrow\ & \texttt{E+id} \\ \rightarrow\ & \texttt{E*E+id} \end{aligned}$$



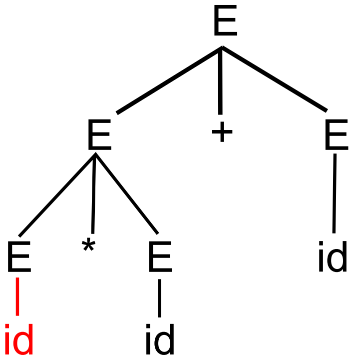
Right-most Derivation in Detail

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



Right-most Derivation in Detail

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



Derivations and Parse Trees

- Observe that left-most and right-most derivations have the same parse tree

Derivations and Parse Trees

- Observe that left-most and right-most derivations have the same parse tree
- The only difference is the order in which branches are added

Derivations and Parse Trees

- Observe that left-most and right-most derivations have the same parse tree
- The only difference is the order in which branches are added
- But when parsing tokens, we only care about the final parse tree, which may have many different derivations

Derivations and Parse Trees

- Observe that left-most and right-most derivations have the same parse tree
- The only difference is the order in which branches are added
- But when parsing tokens, we only care about the final parse tree, which may have many different derivations
- Left-most and right-most derivations are important in parser implementations

Ambiguity

- Recall our example grammar:

Ambiguity

- Recall our example grammar:

```
EXPR → E * E
       | E + E
       | (E)
       | id
```

Ambiguity

- Recall our example grammar:

```
EXPR → E * E
        |  E + E
        |  (E)
        |  id
```

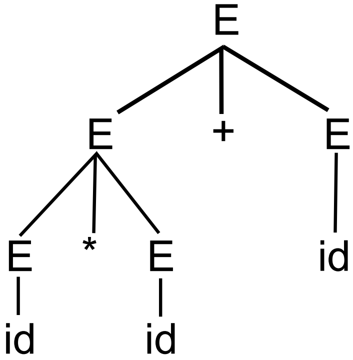
- Now, consider the string `id*id+id`

Ambiguity continued

- This string has two parse trees!

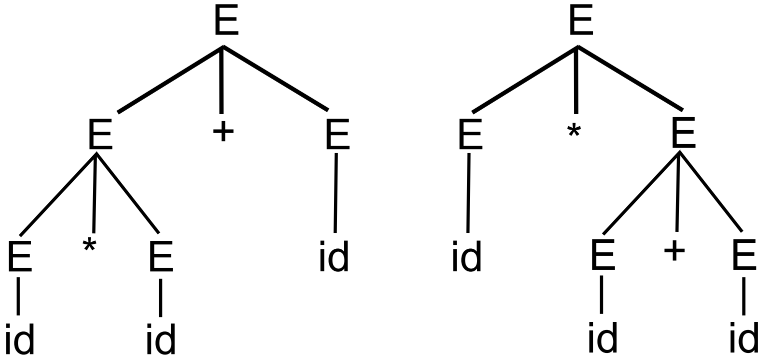
Ambiguity continued

- This string has two parse trees!



Ambiguity continued

- This string has two parse trees!



Ambiguity

- A grammar is ambiguous if it has more than one parse tree for some string

Ambiguity

- A grammar is ambiguous if it has more than one parse tree for some string
- Equivalently: There is more than one left-most or right-most derivation for some string

Ambiguity

- A grammar is ambiguous if it has more than one parse tree for some string
- Equivalently: There is more than one left-most or right-most derivation for some string
- Ambiguity is bad!

Ambiguity

- A grammar is ambiguous if it has more than one parse tree for some string
- Equivalently: There is more than one left-most or right-most derivation for some string
- Ambiguity is bad!
- Leaves meaning of programs ill-defined

Dealing with Ambiguity

- First method: Rewrite grammar unambiguously

Dealing with Ambiguity

- First method: Rewrite grammar unambiguously
- Question: How can we write simple arithmetic expressions unambiguously?

Dealing with Ambiguity

- First method: Rewrite grammar unambiguously
- Question: How can we write simple arithmetic expressions unambiguously?
- Solution: Enforce precedence of times over plus by generating all pluses fist:

Dealing with Ambiguity

- First method: Rewrite grammar unambiguously
- Question: How can we write simple arithmetic expressions unambiguously?
- Solution: Enforce precedence of times over plus by generating all pluses fist:

$$
\begin{array}{lcl}
S & \rightarrow & E + S \mid E \\
E & \rightarrow & \mathsf{id} * E \mid \mathsf{id} \mid (S) * E \mid (S)
\end{array}
$$

Ambiguity

- However, converting grammars to unambiguous form can be very difficult

Ambiguity

- However, converting grammars to unambiguous form can be very difficult
- It also often results in horrible, unintuitive grammars with many non-terminals

Ambiguity

- However, converting grammars to unambiguous form can be very difficult
- It also often results in horrible, unintuitive grammars with many non-terminals
- It is also fundamentally impossible to transform an ambiguous grammar into a unambiguous grammar

Ambiguity

- However, converting grammars to unambiguous form can be very difficult
- It also often results in horrible, unintuitive grammars with many non-terminals
- It is also fundamentally impossible to transform an ambiguous grammar into a unambiguous grammar
- For this reason, tools such as `bison` include disambiguation mechanisms

Precedence and Associativity

- Instead of rewriting the grammar:

Precedence and Associativity

- Instead of rewriting the grammar:
  - Use the more natural ambiguous grammar

Precedence and Associativity

- Instead of rewriting the grammar:
  - Use the more natural ambiguous grammar
  - Along with disambiguating declarations

Precedence and Associativity

- Instead of rewriting the grammar:
  - Use the more natural ambiguous grammar
  - Along with disambiguating declarations
- The parser tool `bison` allows you to declare precedence and associativity for this

Associativity Declarations

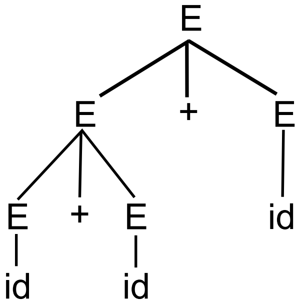
- Consider the grammar $E \rightarrow E + E \mid \text{id}$

Associativity Declarations

- Consider the grammar $E \rightarrow E + E \mid \text{id}$
- Ambiguous: Two parse trees of input `id + id + id`

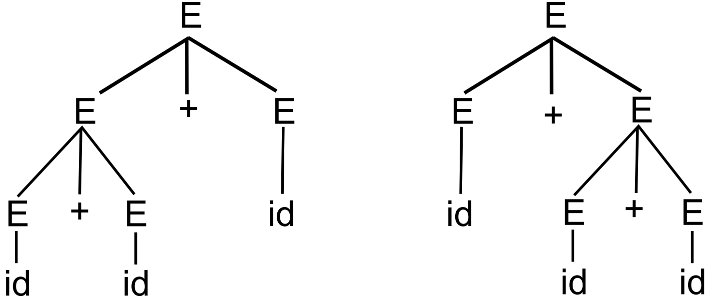
Associativity Declarations

- Consider the grammar $E \rightarrow E + E \mid \text{id}$
- Ambiguous: Two parse trees of input `id + id + id`



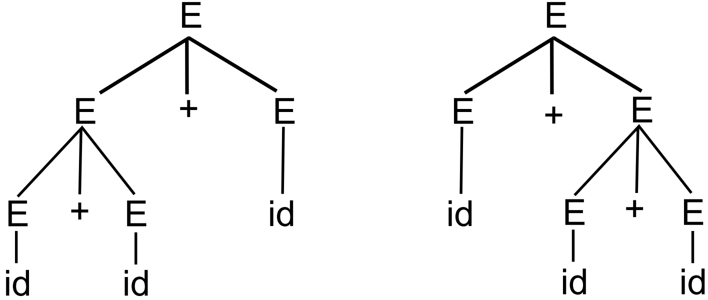
Associativity Declarations

- Consider the grammar $E \to E + E \mid \text{id}$
- Ambiguous: Two parse trees of input `id + id + id`



Associativity Declarations

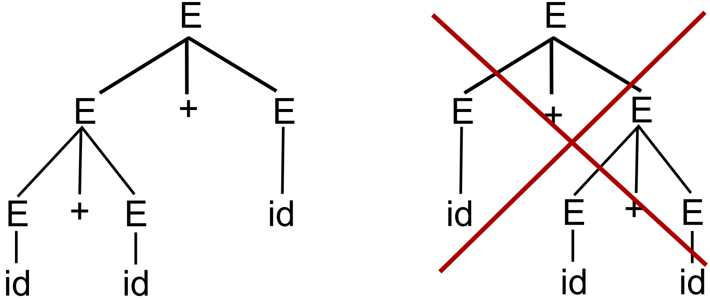
- Consider the grammar $E \to E + E \mid \text{id}$
- Ambiguous: Two parse trees of input `id + id + id`



- Declare left associativity of plus as: `%left +`

Associativity Declarations

- Consider the grammar $E \to E + E \mid \text{id}$
- Ambiguous: Two parse trees of input `id + id + id`



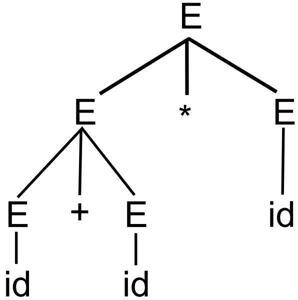
- Declare left associativity of plus as: `%left +`

Precedence Declarations

- Consider the grammar $E \rightarrow E + E \mid \text{id}$ and input `id + id * id`

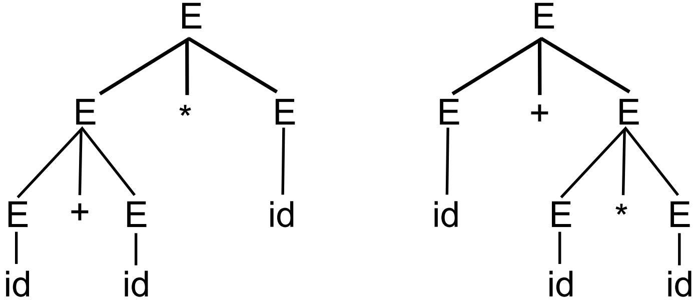
Precedence Declarations

- Consider the grammar $E \rightarrow E + E \mid \text{id}$ and input `id + id * id`



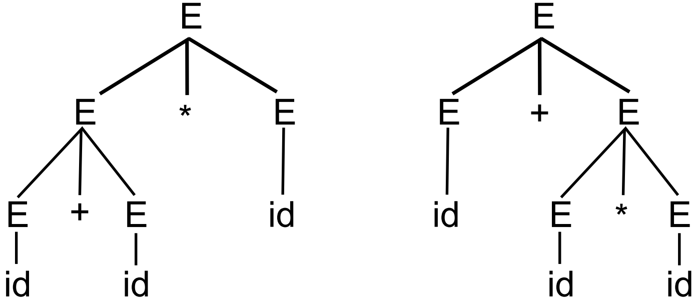
Precedence Declarations

- Consider the grammar $E \rightarrow E + E \mid \text{id}$ and input `id + id * id`



Precedence Declarations

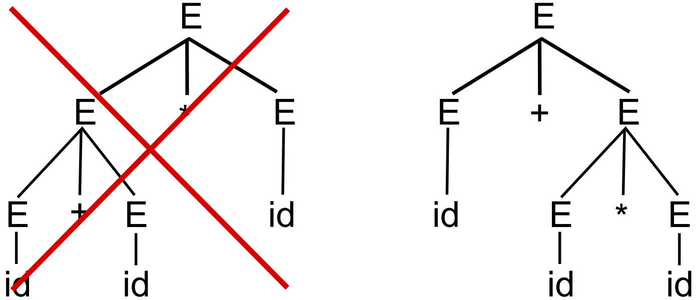
- Consider the grammar $E \rightarrow E + E \mid \text{id}$ and input `id + id * id`



- Precedence Declaration:
  `%left +`
  `%left *`

Precedence Declarations

- Consider the grammar $E \rightarrow E + E \mid \text{id}$ and input `id + id * id`



- Precedence Declaration:
  `%left +`
  `%left *`

Conclusion

- We have seem how to specify programming language syntax with CFGs

Conclusion

- We have seem how to specify programming language syntax with CFGs
- We built parse trees that express the high-level syntactic structure

Conclusion

- We have seem how to specify programming language syntax with CFGs
- We built parse trees that express the high-level syntactic structure
- Parse trees of programs are known as abstract syntax trees

Conclusion

- We have seem how to specify programming language syntax with CFGs
- We built parse trees that express the high-level syntactic structure
- Parse trees of programs are known as abstract syntax trees
- We discussed ambiguity of CFGs