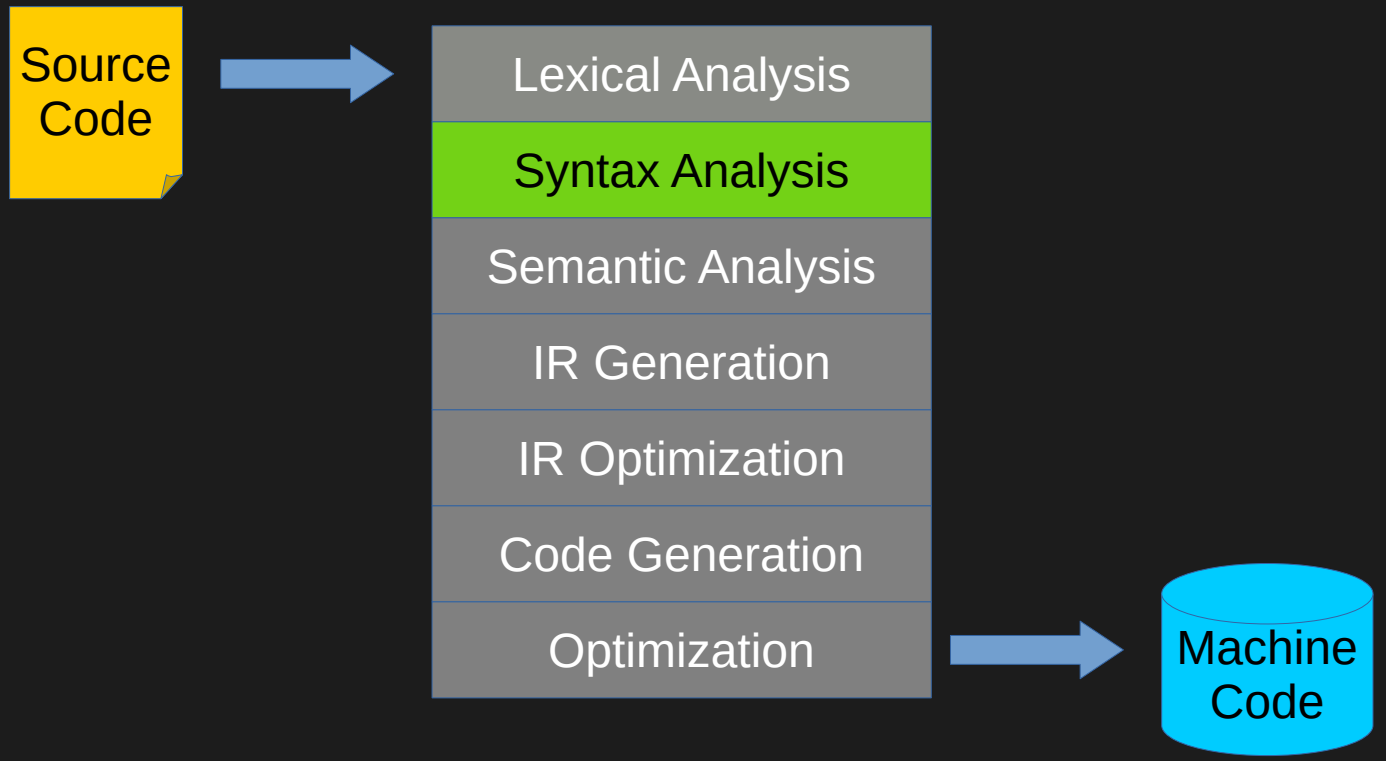
Syntax Analysis – Grammars

Wei Wang

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Textbook Chapters

- Dragon book
  - Chapter 4.1, 4.2 and 4.3

What is Syntax Analysis?

- After lexical analysis (scanning), we have a series of tokens.
- In **syntax analysis** (or **parsing**), we want to interpret what those tokens mean.
- Goal: Recover the ***structure*** described by that series of tokens.
- Goal: Report ***errors*** if those tokens do not properly encode a structure.

Context-free Grammars

Lectures in Syntax Analysis

- Context-free grammars
  - Derivations
  - Syntax Trees
  - Ambiguity
- Parsing algorithms
  - Top-down parsing
  - Bottom-up parsing

Formal Languages

- Recall that,
  - An **alphabet** is a set Σ of symbols that act as letters.
  - A **language** over Σ is a set of strings made from symbols in Σ.
- When scanning, our alphabet was ASCII or Unicode characters. We produced tokens.
- When parsing, our alphabet is the set of tokens produced by the scanner.

The Limits of Regular Languages

- When scanning, we used regular expressions to define each token.
- Unfortunately, regular expressions are (usually) too weak to define programming languages.
  - Cannot define a regular expression matching all expressions with properly balanced parentheses.
  - Cannot define a regular expression matching all functions with properly nested block structure.
- We need a more powerful formalism.

Context-Free Grammars

- A context-free grammar (or CFG) is a formalism for defining languages.
- Recall the definition of CFG:
  - For a production in CFG $\beta \rightarrow \alpha$,
    - β must contain only one non-terminal symbols
    - α may contain one or more terminal and non-terminal symbols, or α may be the empty character ε.

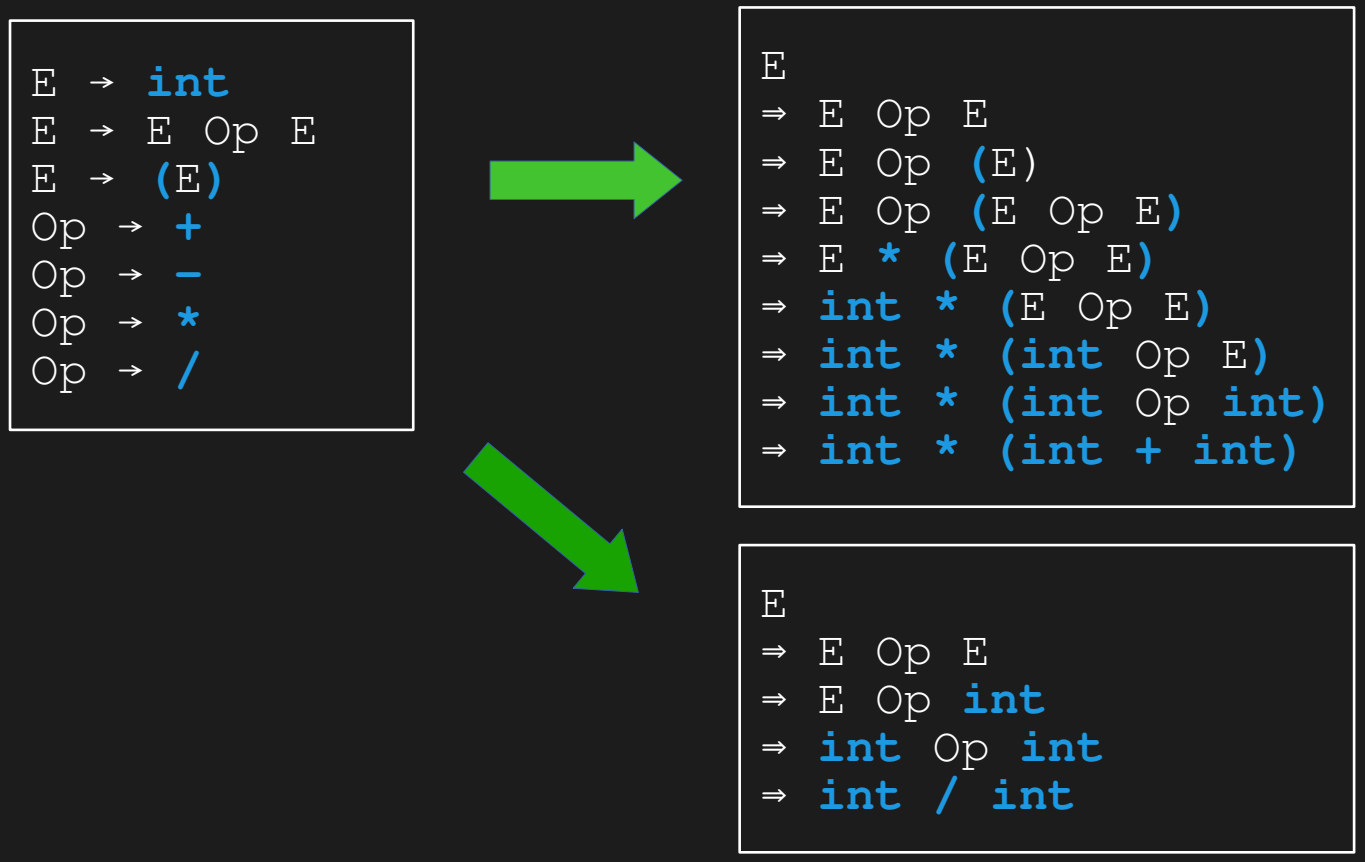
An Example of CFG

- Arithmetic Expressions:
  - Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
  - The CFG may be (blue symbols are terminal):

```
E → int
E → E Op E
E → (E)
Op → +
Op → -
Op → *
Op → /
```

An Example of CFG cont.

- With this CFG, we can get:



A Notational Shorthand

- To make it easier for implementation, we can merge similar productions:
  - Use "|" to separate rules with the same left hand.

```
E → int
E → E Op E
E → (E)
Op → +
Op → -
Op → *
Op → /
```

```
E → int | E Op E | (E)
Op → + | - | * | /
```

A Note on CFG

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, + or parentheses as special characters.

Some CFG Notations

- Capital letters at the beginning of the alphabet will represent nonterminals.
  - i.e. **A, B, C, D**
- Lowercase letters at the end of the alphabet will represent terminals.
  - i.e. **t, u, v, w**
- Lowercase Greek letters will represent arbitrary strings of terminals and nonterminals.
  - i.e. **α, γ, ω**

Examples

- We might write an arbitrary production as
  - A → ω
- We might write a string of a nonterminal followed by a terminal as
  - At
- We might write an arbitrary production containing a nonterminal followed by a terminal as
  - B → αAtω

Example: CFGs for Programming Languages

```
BLOCK → STMT
      | { STMTS }
STMTS → ε
      | STMT STMTS
STMT → EXPR;
      | if (EXPR) BLOCK
      | while (EXPR) BLOCK
      | do BLOCK while (EXPR);
      | BLOCK
      | …
EXPR → identifier
      | constant
      | EXPR + EXPR
      | EXPR – EXPR
      | EXPR * EXPR
      | ...
```

Derivations

Derivations

- This sequence of steps is called a derivation.
- A string $\alpha A \omega$ yields string $\alpha \gamma \omega$ iff $A \rightarrow \gamma$ is a production.
- If α yields β, we write $\alpha \Rightarrow \beta$.
- We say that α derives β *iff* there is a sequence of strings where

  $\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \beta$
- If α derives β, we write $\alpha \Rightarrow^* \beta$.

Example: Leftmost Derivations

Grammars:

```
BLOCK → STMT
      | { STMTS }
STMTS → ε
      | STMT STMTS
STMT → EXPR;
      | if (EXPR) BLOCK
      | while (EXPR) BLOCK
      | do BLOCK while (EXPR);
      | BLOCK
      | …
EXPR → identifier
      | constant
      | EXPR + EXPR
      | EXPR - EXPR
      | EXPR * EXPR
      | EXPR = EXPR
      | ...
```

Derivations:

```
STMTS
⇒ STMT STMTS
⇒ EXPR; STMTS
⇒ EXPR = EXPR; STMTS
⇒ id = EXPR; STMTS
⇒ id = EXPR + EXPR; STMTS
⇒ id = id + EXPR; STMTS
⇒ id = id + constant; STMTS
⇒ id = id + constant;
```

Leftmost and Rightmost Derivations

- A leftmost derivation is a derivation in which each step expands the leftmost nonterminal.
- A rightmost derivation is a derivation in which each step expands the rightmost nonterminal.

Left- vs Right-most Derivations

Left-most derivation:

```
E
⇒ E Op E
⇒ int Op E
⇒ int * E
⇒ int * (E)
⇒ int * (E Op E)
⇒ int * (int Op E)
⇒ int * (int + E)
⇒ int * (int + int)
```

Right-most derivation:

```
E
⇒ E Op E
⇒ E Op (E)
⇒ E Op (E Op E)
⇒ E Op (E Op int)
⇒ E Op (E + int)
⇒ E Op (int + int)
⇒ E * (int + int)
⇒ int * (int + int)
```

- These two derivations are basically equivalent, although they may affect how parsing is implemented.

The Importance of Derivations

- A derivation encodes two pieces of information:
  - What productions were applied to produce the resulting string from the start symbol?
  - In what order were they applied?
- Multiple derivations might use the same productions, but apply them in a different order.

Parse Trees

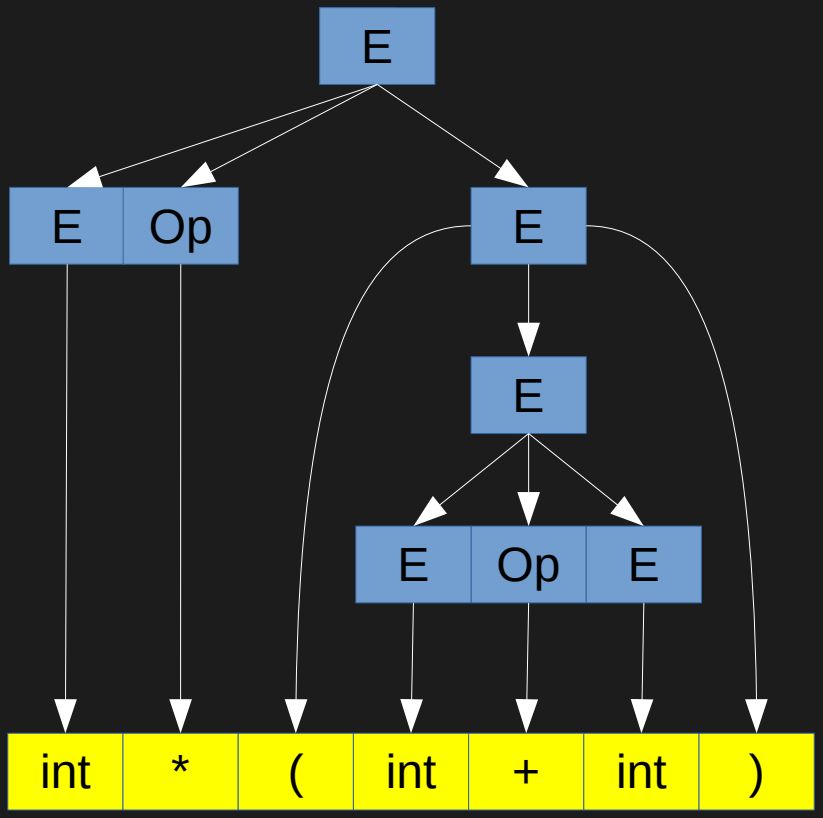
Parse Trees

- A parse tree is a tree encoding the steps in a derivation.
- Internal nodes represent nonterminal symbols used in the production.
- In-order walk of the leaves contains the generated string.
- Encodes what productions are used, not the order in which those productions are applied.

An Example of Parse Tree

```
E
⇒ E Op E
⇒ int Op E
⇒ int * E
⇒ int * (E)
⇒ int * (E Op E)
⇒ int * (int Op E)
⇒ int * (int + E)
⇒ int * (int + int)
```

```
E
⇒ E Op E
⇒ E Op (E)
⇒ E Op (E Op E)
⇒ E Op (E Op int)
⇒ E Op (E + int)
⇒ E Op (int + int)
⇒ E * (int + int)
⇒ int * (int + int)
```



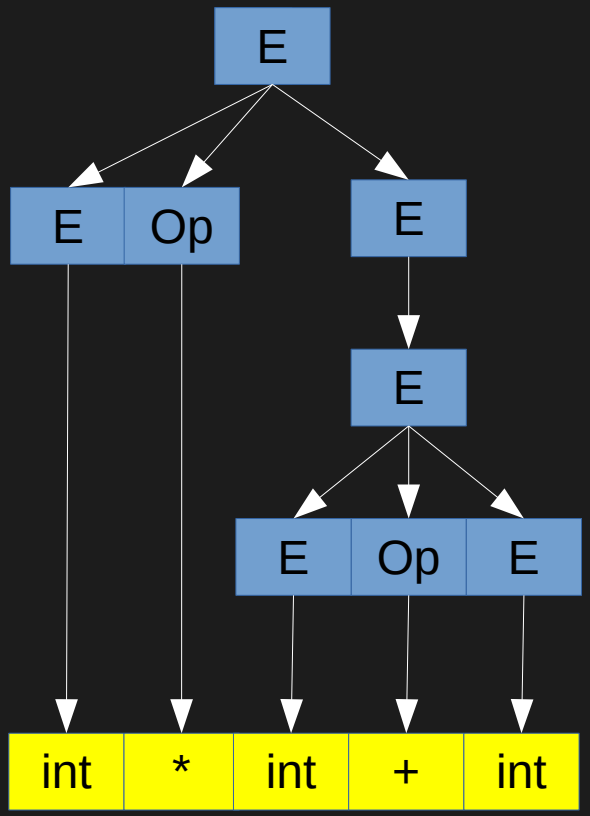
Note that although both left- and right-most derivations
gives the same pass tree, the exact tree generating processes are different.

The Goal of Parsing

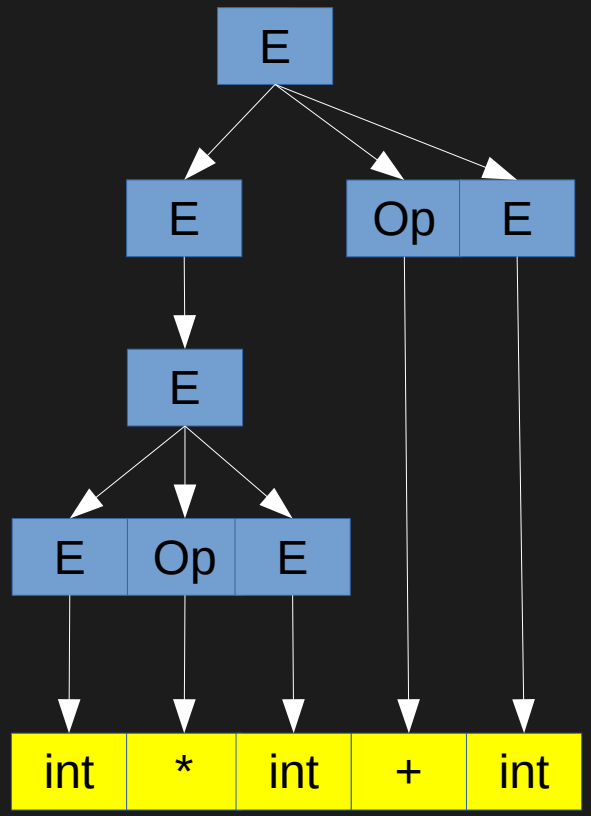
- Goal of syntax analysis: recover the structure described by a series of tokens.
- If language is described as a CFG, goal is to recover a parse tree for the the input string.
- Usually we do some simplifications on the tree; more on that later.
- We'll discuss how to do this next week.

Challenges in Parsing

A Serious Problem



int * (int + int)



(int * int) + int

Ambiguity

- A CFG is said to be ambiguous if there is at least one string with two or more parse trees.
- Note that ambiguity is a property of *grammars*, not *languages*.
- There is no algorithm for converting an arbitrary ambiguous grammar into an unambiguous one.
  - Some languages are inherently ambiguous, meaning that no unambiguous grammar exists for them.
- There is no algorithm for detecting whether an arbitrary grammar is ambiguous.

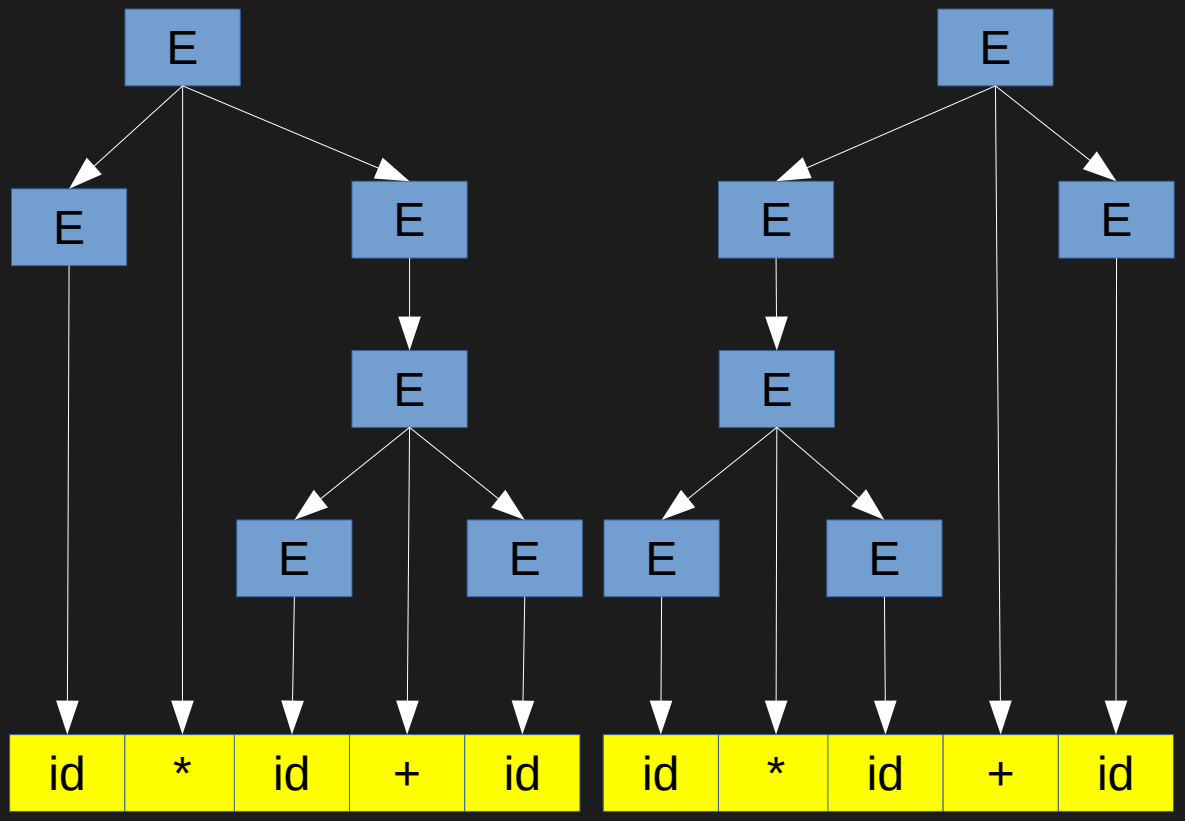
Resolving Ambiguity

- If a grammar can be made unambiguous at all, it is usually made unambiguous through layering.
  - Have exactly one way to build each piece of the string.
  - Have exactly one way of combining those pieces back together.

An Example of Layering

- Consider this ambiguous grammar:

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



int * (int + int)

(int * int) + int

An Example of Layering cont.

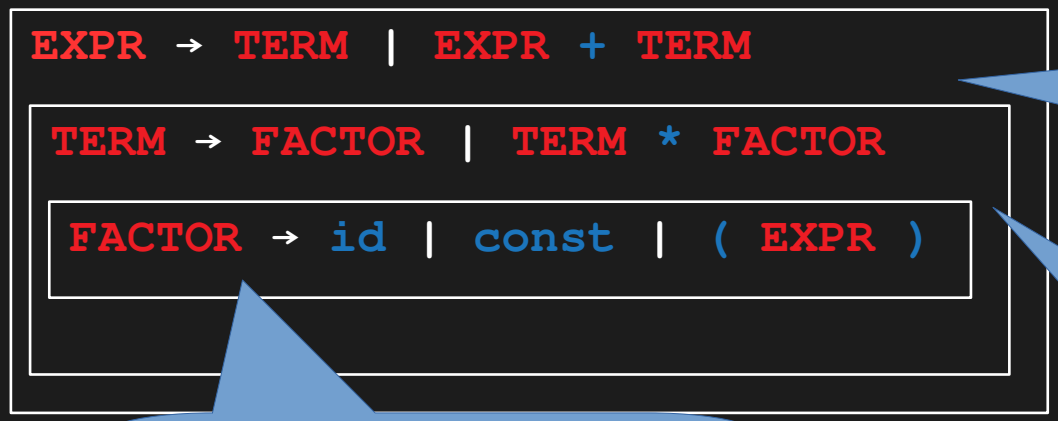
- We can try to resolve the ambiguity via layering:
  - Essentially break the EXPR symbol into multiple layers of non-terminal symbols based on precedence and associativity.

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

```
EXPR → TERM | EXPR + TERM
TERM → FACTOR | TERM * FACTOR
FACTOR → id | const | ( EXPR )
```

An Example of Layering cont.

- Explanation for the new grammar:



EXPR is used to represent addition which has lower precedence than multiplication.
E + T also specifies + is left-associative.

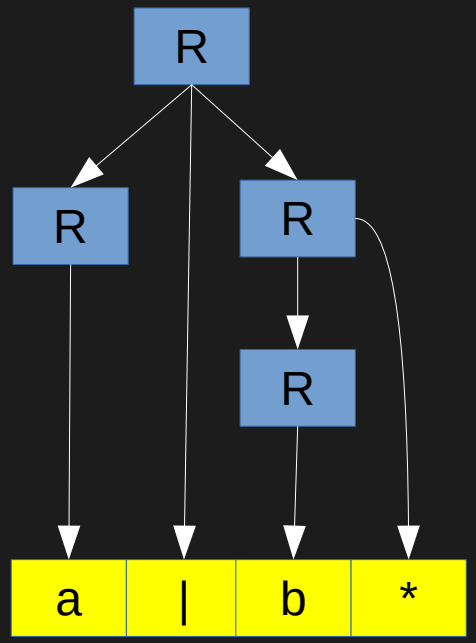
FACTOR is the atomic component of an expression.

TERM is used to represent multiplication which has higher precedence than addition.
T * F also specifies * is left-associative.

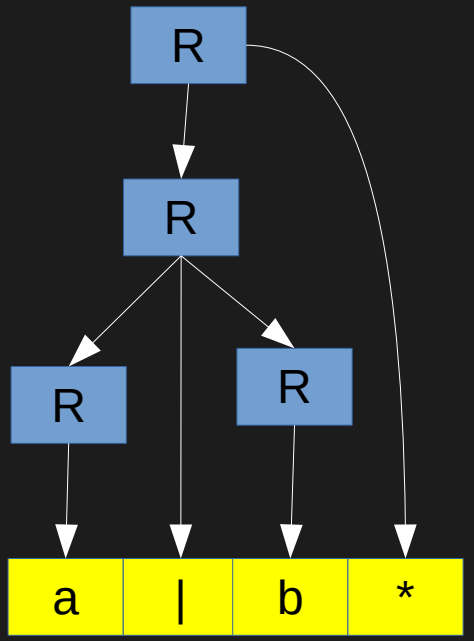
Another Example of Layering

- Consider the following ambiguous grammars that describes regular expressions.

```
R → a | b | c | …
R → “ε”
R → RR
R → R “|” R
R → R*
R → (R)
```



a | (b*)



(a | b)*

Another Example of Layering cont.

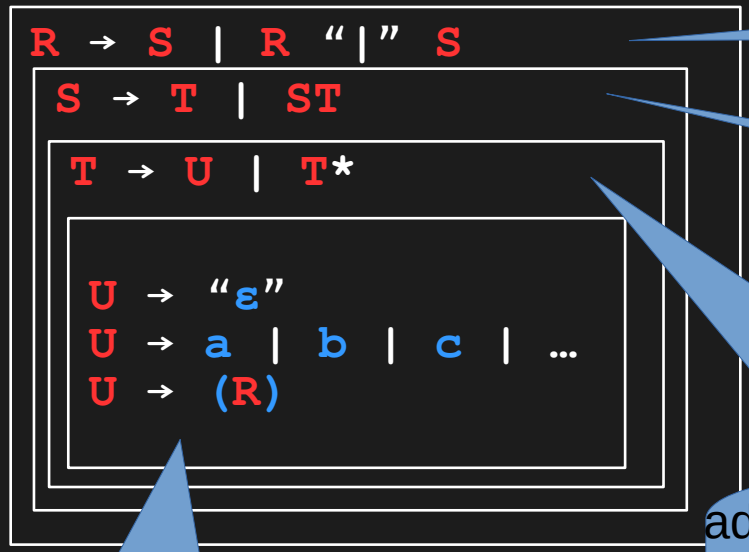
- We can try to resolve the ambiguity via layering:
  - Essentially break the R symbol into multiple layers of non-terminal symbols based on precedence and associativity.

```
R → a | b | c | …
R → “ε”
R → RR
R → R “|” R
R → R*
R → (R)
```

```
R → S | R “|” S
S → T | ST
T → U | T*
U → “ε”
U → a | b | c | …
U → (R)
```

Another Example of Layering cont.

- Why the new grammar is not ambiguous?



Unions concatenated expressions on top showing lowest precedence

Concatenates starred Expressions to allow proper generation of unary operation "star" is left-associative

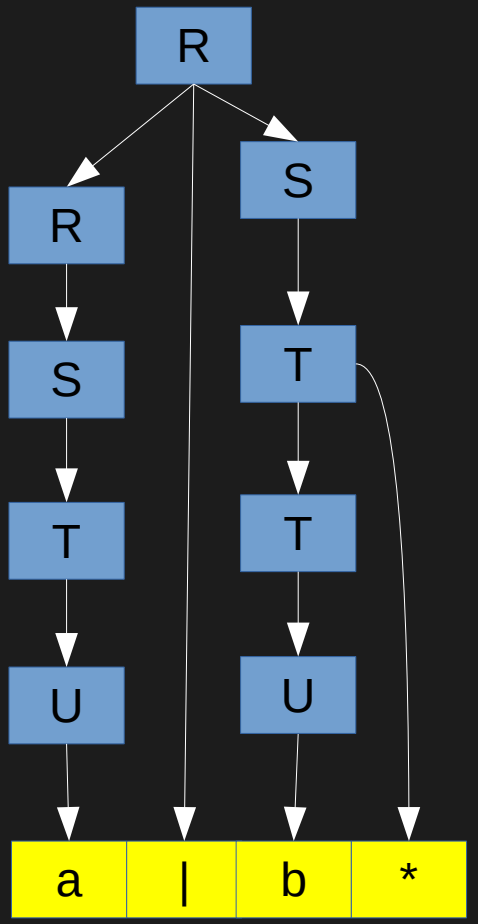
add an extra Non-Term Symbol to put stars onto atomic expressions to show that it has the highest precedence

Only generates "atomic" expressions

Another Example of Layering cont.

- The only parse tree with the new grammar:

```
R → S | R "|" S
S → T | ST
T → U | T*
U → "ε"
U → a | b | c | …
U → (R)
```



a | (b*)

Precedence Declarations

- If we leave the world of pure CFGs, we can often resolve ambiguities through precedence declarations.
  - e.g. multiplication has higher precedence than addition, but lower precedence than exponentiation.
  - Associativity is also important!
- Allows for unambiguous parsing of ambiguous grammars.
- Most of the time, we use precedence declarations to resolve ambiguity.
  - In Yacc and Bison, we can simply declare that multiplication has higher precedence than addition, and it is also left-associative.

Abstract Syntax Trees

Abstract Syntax Trees (ASTs)

- A parse tree is a concrete syntax tree; it shows exactly how the text was derived.
- A more useful structure is an abstract syntax tree, which retains only the essential structure of the input.
  - Invented when memory space was limited.

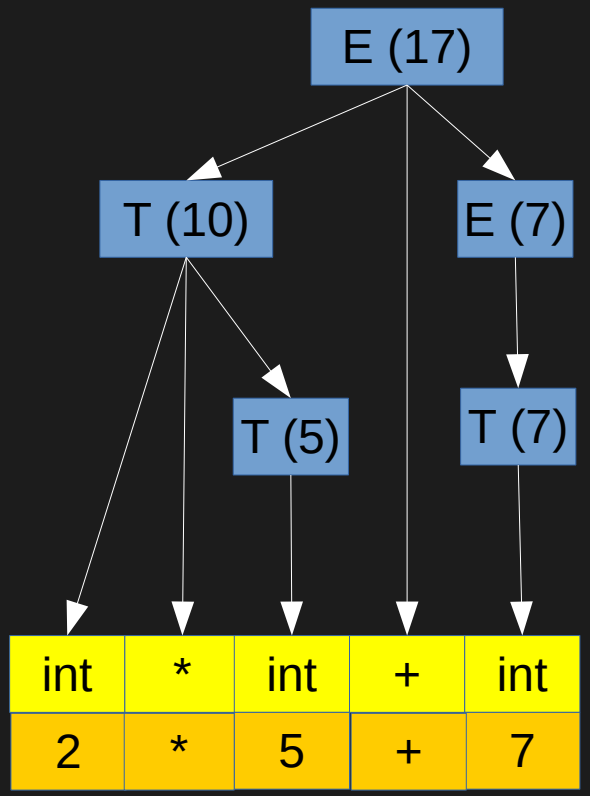
How to build an AST?

- Typically done through semantic actions.
- Associate a piece of code to execute with each production.
- As the input is parsed, execute this code to build the AST.
  - Exact order of code execution depends on the parsing method used.
- This is called a syntax-directed translation.
- Project 2 asks you to write grammars and build an AST with semantic actions.

Simple Semantic Actions : A Calculator

- The following grammar and its associated semantic actions can be used to compute simple expressions.

E → T + E	E_1.val = T.val + E_2.val
E → T	E.val = T.val
T → int	T.val = int.val
T → int * T	T.val = int.val * T.val
T → (E)	T.val = E.val



*Note a bottom-up parser is required to carry out these operations.

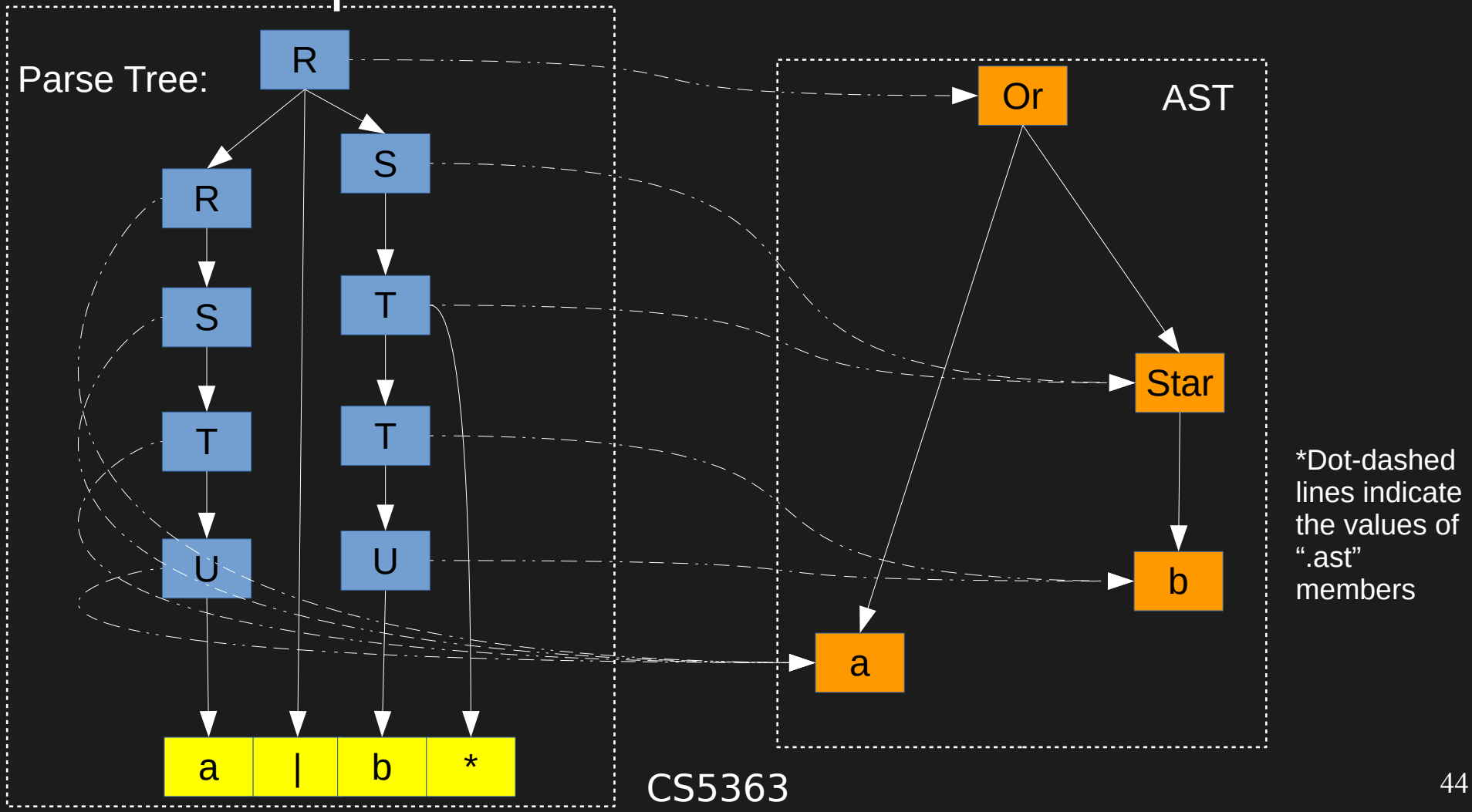
An Example of Semantic Actions to Build ASTs

- Consider the regular expression grammar in previous slides.

R → S	R.ast = S.ast;
R → R "|" S	R_1.ast = new Or(R_2.ast, S.ast);
S → T	S.ast = T.ast;
S → ST	S_1.ast = new Concat(S_2.ast, T.ast);
T → U	T.ast = U.ast;
T → T*	T.ast = new Star(T_2.ast);
U → a	U.ast = new SingleChar('a');
U → b	U.ast = new SingleChar('b');
U → "ε"	U.ast = new Epsilon();
U → (R)	U.ast = R.ast;

An Example of Semantic Actions to Build ASTs cont.

- An example AST:



Summary

- Syntax analysis (parsing) extracts the structure from the tokens produced by the scanner.
- Languages are usually specified by context-free grammars (CFGs).
- A parse tree shows how a string can be derived from a grammar.
- A grammar is ambiguous if it can derive the same string multiple ways.
- There is no algorithm for eliminating ambiguity; it must be done by hand.
- Abstract syntax trees (ASTs) contain an abstract representation of a program's syntax.
- Semantic actions associated with productions can be used to build ASTs.

Recognizing CFGs

Recognizing CFGs

- Similar to regular expressions in lexical analysis, syntax analysis requires an algorithm to recognize the strings from a context-free language.
- Similar to DFA/NFA for RE, Pushdown Automaton (PDA) is the theoretical machine model for recognizing the language specified by a CFG.

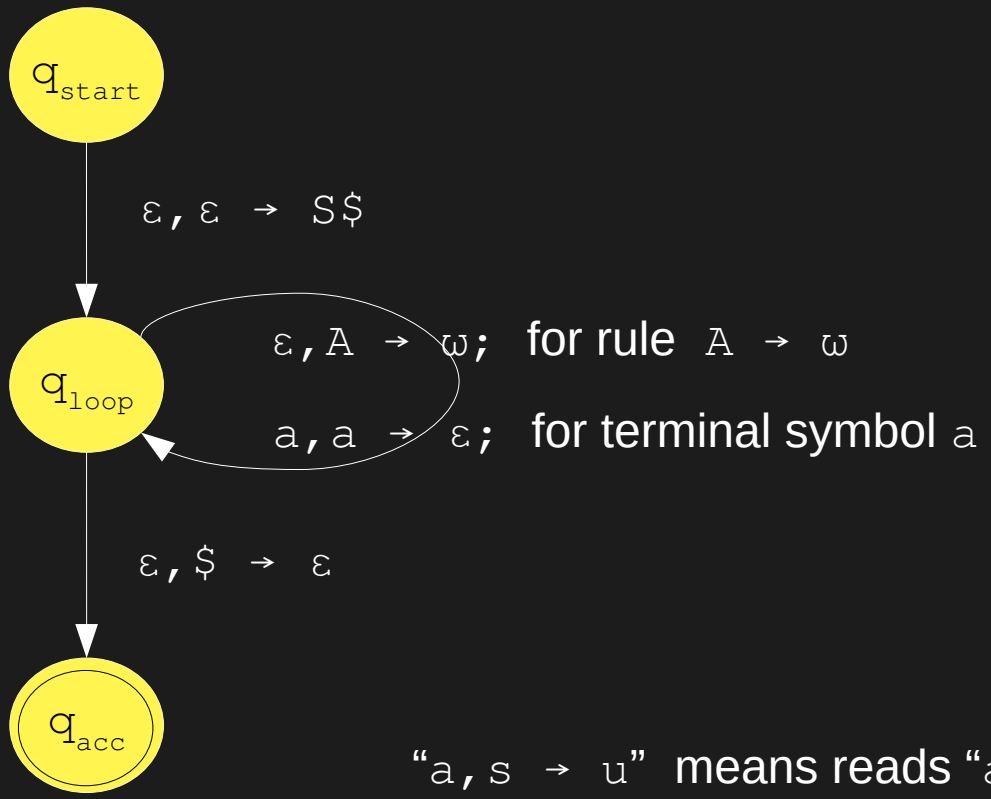
Pushdown Automaton

- CFG and PDA are equivalent in power: a CFG generates a context-free language and a PDA recognizes a context-free language.
- A PDA is essentially a state machine with a stack (last-in first-out).
  - As a comparison, DFA/NFA are just state machines.
  - Beside traveling among states, the transitions in PDA also specify the operations performed on the stack.

Constructing PDA for a CFG

- A PDA for a CFG has only three states:
  - The states are $Q = \{q_{start}, q_{loop}, q_{accept}\}$.
- The first transition travels from q_{start} to q_{loop}, when reading ε.
  - This transition also puts "$" and "S" on to the stack
  - "$" is viewed as a special character indicating the end of the stack.
  - "S" is the starting symbol of the CFG.
- q_{loop}, as its name suggests, loops at itself, with three transitions.
  - If the top of the stack is a non-terminal symbol N, non-deterministically pick a production that has N on the left hand, and replace N in the stack with the right hand of the production.
  - If the top of the stack is a terminal symbol T, reads one character C from the input string. If (T==C), pop T from the stack; otherwise, reject.
  - If the top of the stack is "$" and the input string is empty, go to q_{accept} (accept). If the top of the stack is "$", but the input string is not empty, reject.

State Diagram of a PDA



"a,s → u" means reads "a" from input string, pops "s" from the stack, and push "u" on the stack.

The Problem of Non-deterministic PDAs

- It is possible to implement non-deterministic PDA by tracking each branch of executions.
  - However, this implementation is very inefficient.
- For certain CFGs, it is possible to construct deterministic PDAs.
  - These CFGs are called deterministic CFGs.
- However, there is no generic algorithm to convert a non-deterministic PDA to a deterministic one.
  - Actually, some CFGs do not have deterministic PDAs.
- Luckily, the CFGs of common programming languages are all deterministic. There are several algorithms to construct deterministic PDAs for them, which are the main topics for the next few lectures.

Acknowledgement

- This lectures is partially based on the Compiler slides of Keith Schwarz, partially based on the Theory slides of Dr. Michael A. Bender.