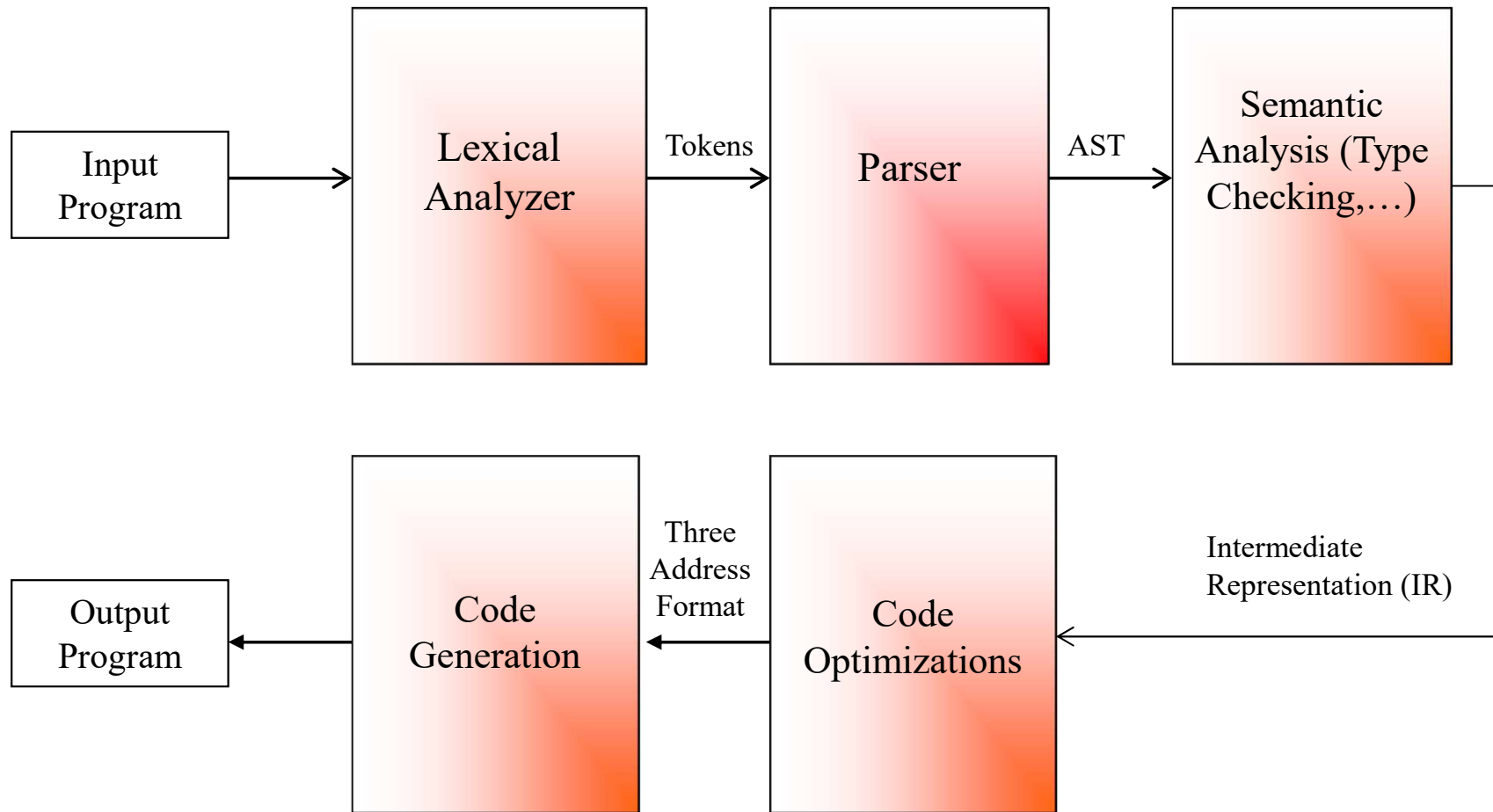


Introduction to Parsers and Recursive Descent Parsing

Lecture Outline

- So far in the course, we have covered basics of computability and formal languages, regular languages and their properties, context-free languages/grammars and their properties, lexers
- Introduction to parsing
- Recursive descent
- Extensions of CFG for parsing
 - Precedence declarations
 - Error handling
 - Semantic actions
- Constructing a parse tree

The Structure (Phases) of a Compiler



Derivations and Parse Trees

A *derivation* is a sequence of productions

$$S \rightarrow \dots \rightarrow \dots \rightarrow \dots$$

A derivation can be drawn as a tree

- Start symbol is the tree's root

- For a production $X \rightarrow Y_1 \cdots Y_n$

- add children $Y_1 \cdots Y_n$ to node X

Derivation Example

- Grammar

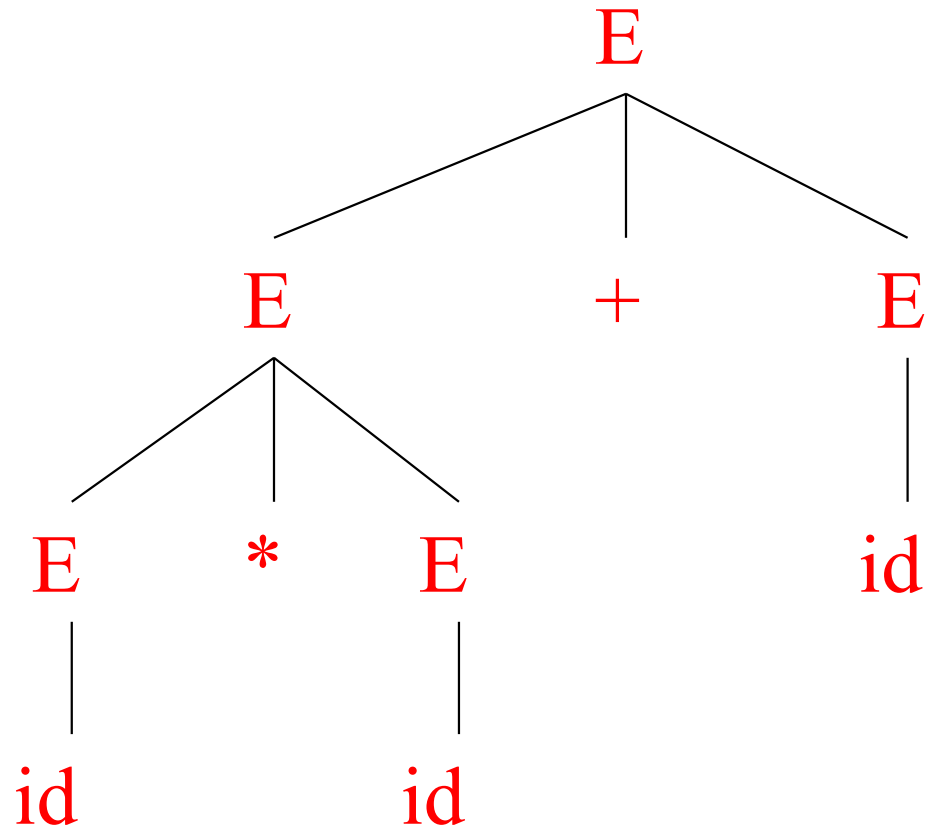
$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

- String

$$id * id + id$$

Derivation Example (Cont.)

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



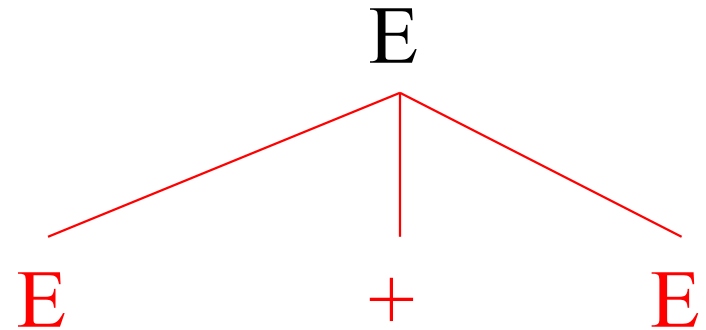
Derivation in Detail (1)

E

E

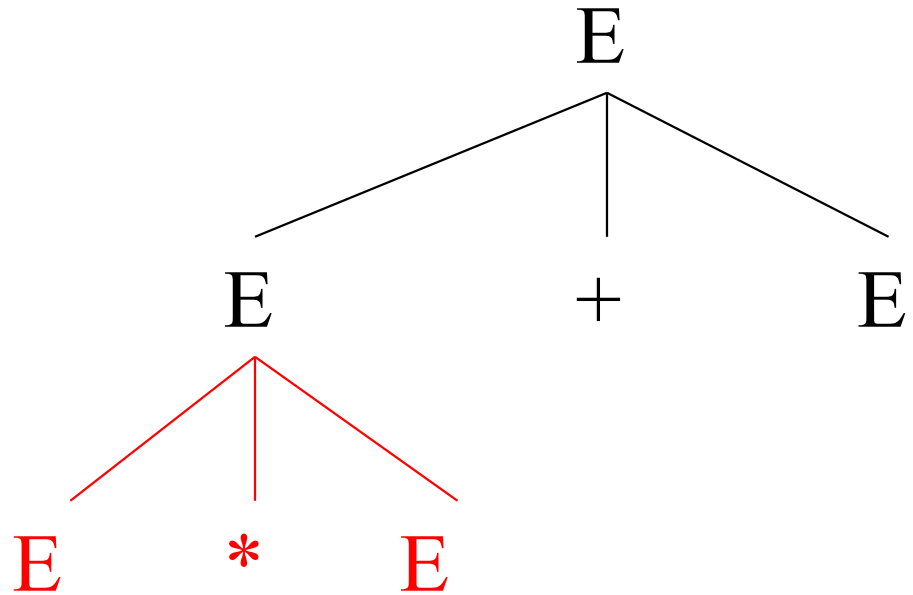
Derivation in Detail (2)

E
 $\rightarrow E + E$



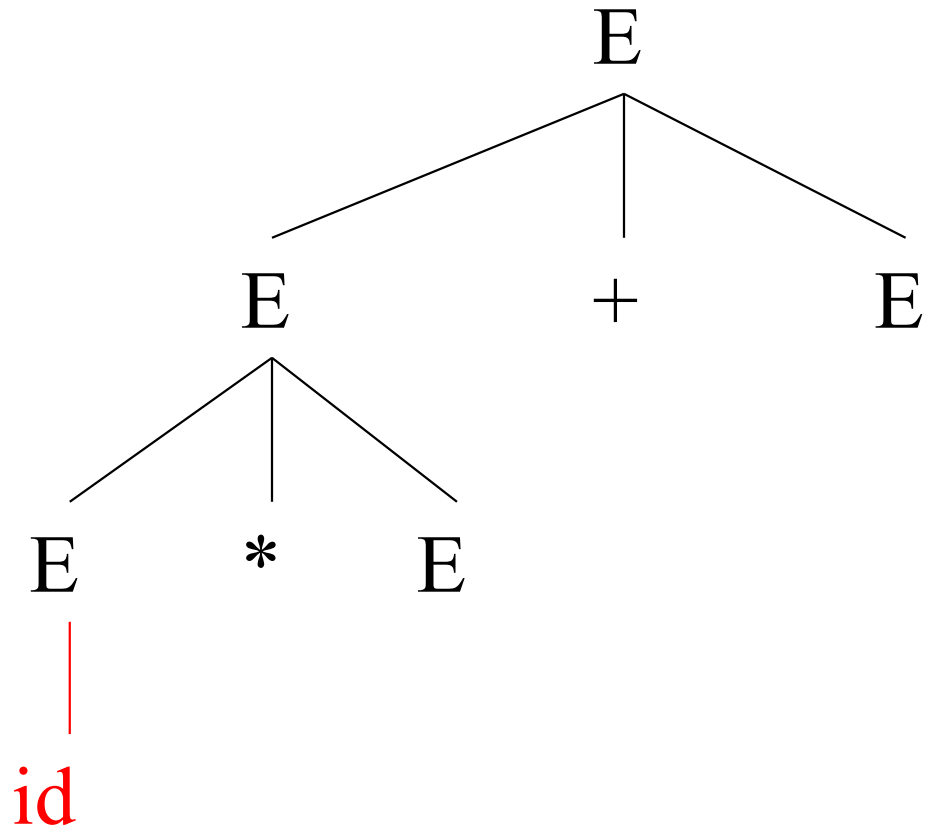
Derivation in Detail (3)

E
 $\rightarrow E + E$
 $\rightarrow E * E + E$



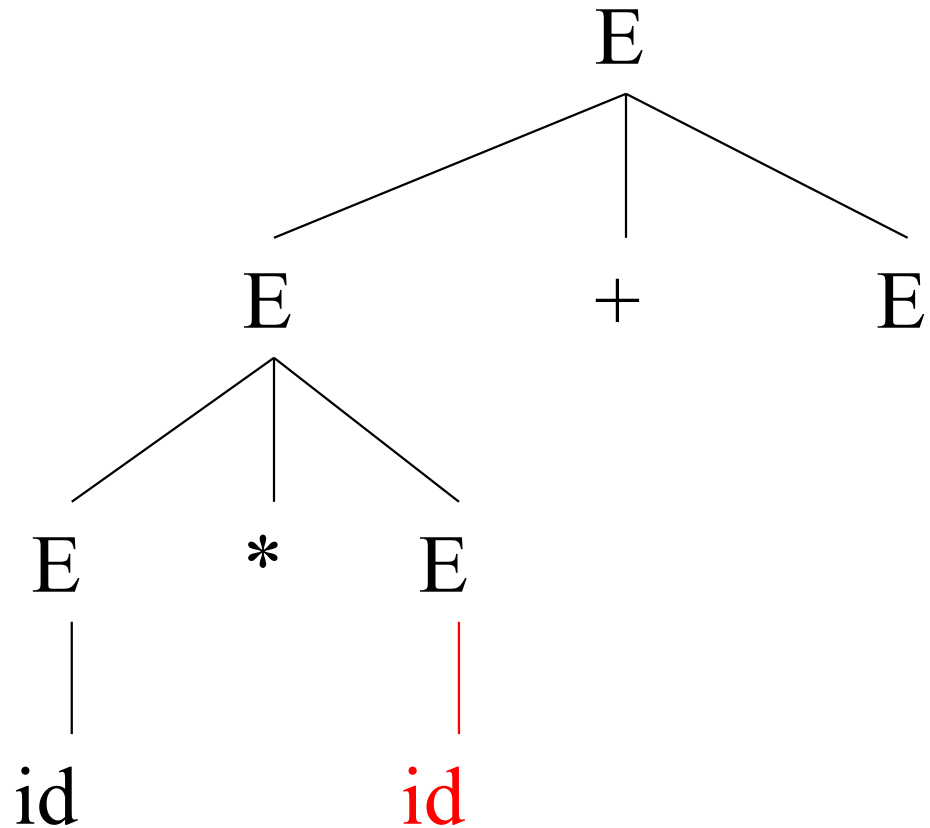
Derivation in Detail (4)

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



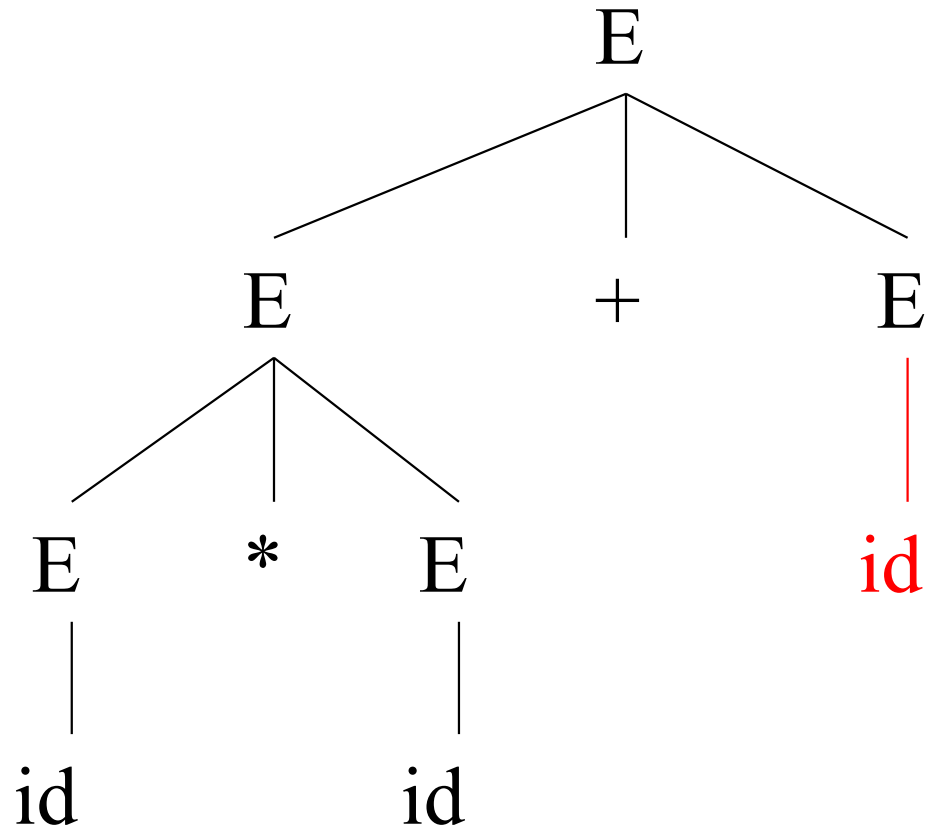
Derivation in Detail (5)

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



Derivation in Detail (6)

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



Notes on Derivations

- A parse tree has
 - Terminals at the leaves
 - Non-terminals at the interior nodes
- An in-order traversal of the leaves is the original input
- The parse tree shows the association of operations, the input string does not

Left-most and Right-most Derivations

- The example is a *left-most* derivation
 - At each step, replace the left-most non-terminal
- There is an equivalent notion of a *right-most* derivation

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

Bottom-up vs. Top-down

	Top-down Parsers	Bottom-up Parsers
Successful Parse	From start symbol of grammar to the string	From the string to the start symbol of the grammar
Example of grammars	LL(k) Left-to-right, Leftmost derivation first	LR(k), LALR Left-to-right, Rightmost derivation first (in reverse)
Example of parser technique	Recursive-descent	Shift-reduce
Ease of implementation	Very easy	Many grammar generators available (Yacc, Bison,...)
Issue with left-recursion	Yes	No
Issues with left-factoring	Yes	No

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

E

(int₅)



Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

E
|
 T

(int_5)



Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

E
|
T
|
int

*Mismatch: int is not (!
Backtrack ...*

(int₅)



Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

E
|
 T

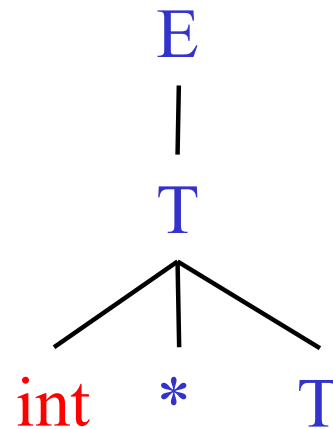
(int_5)



Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



*Mismatch: int is not (!
Backtrack ...*

(int₅)



Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

E
|
 T

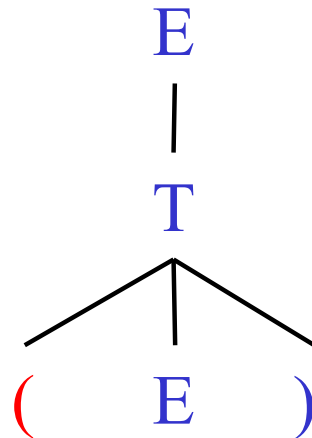
(int_5)



Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



Match! Advance input.

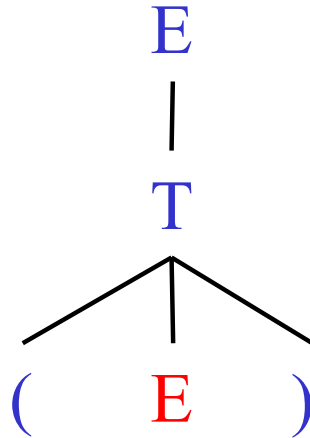
(int_5)



Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

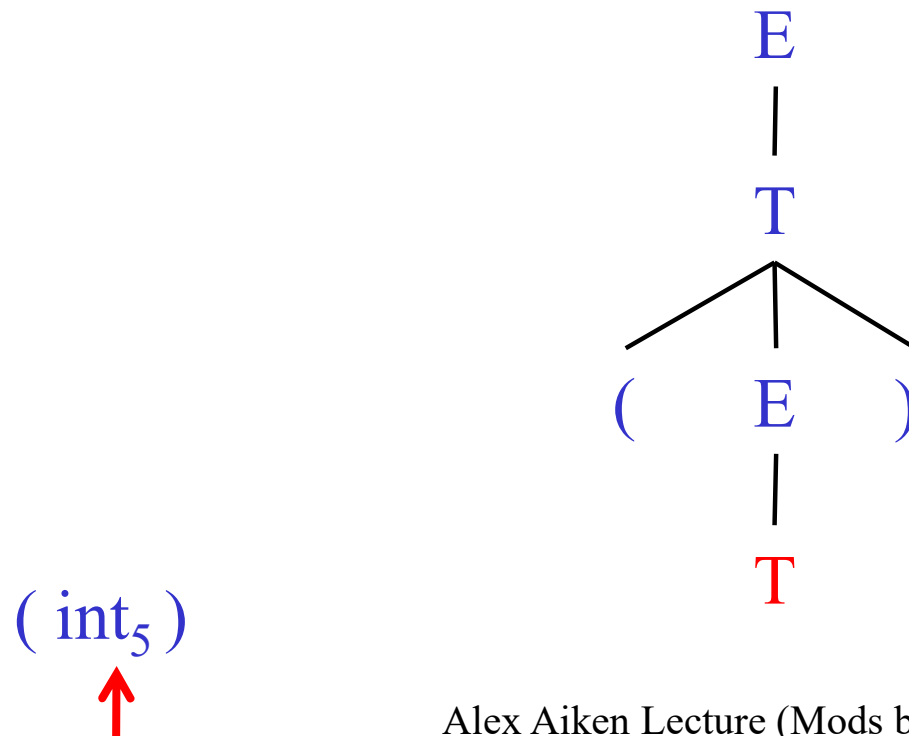


(int_5)
↑

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

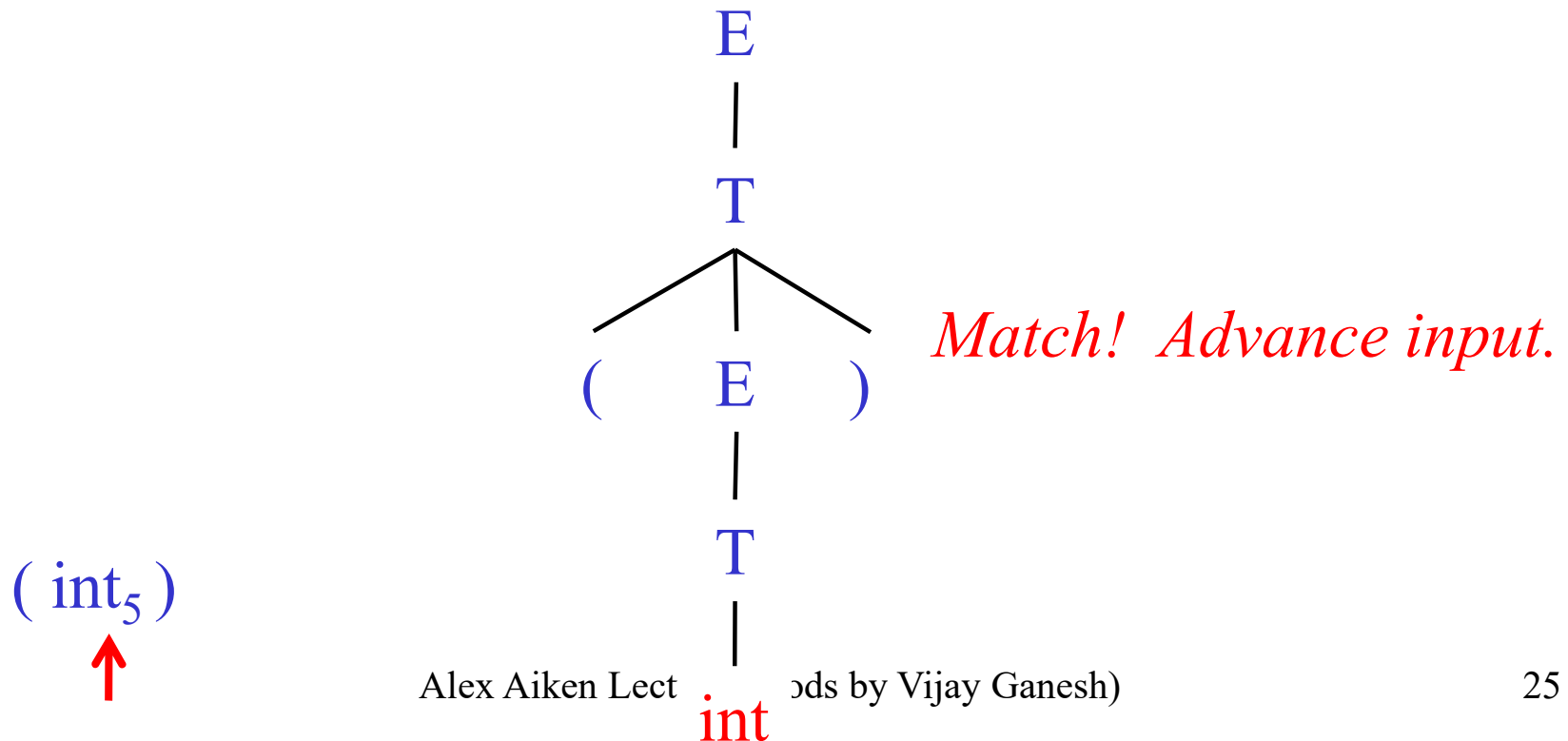
$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



Recursive Descent Parsing

$E \rightarrow T \mid T + E$

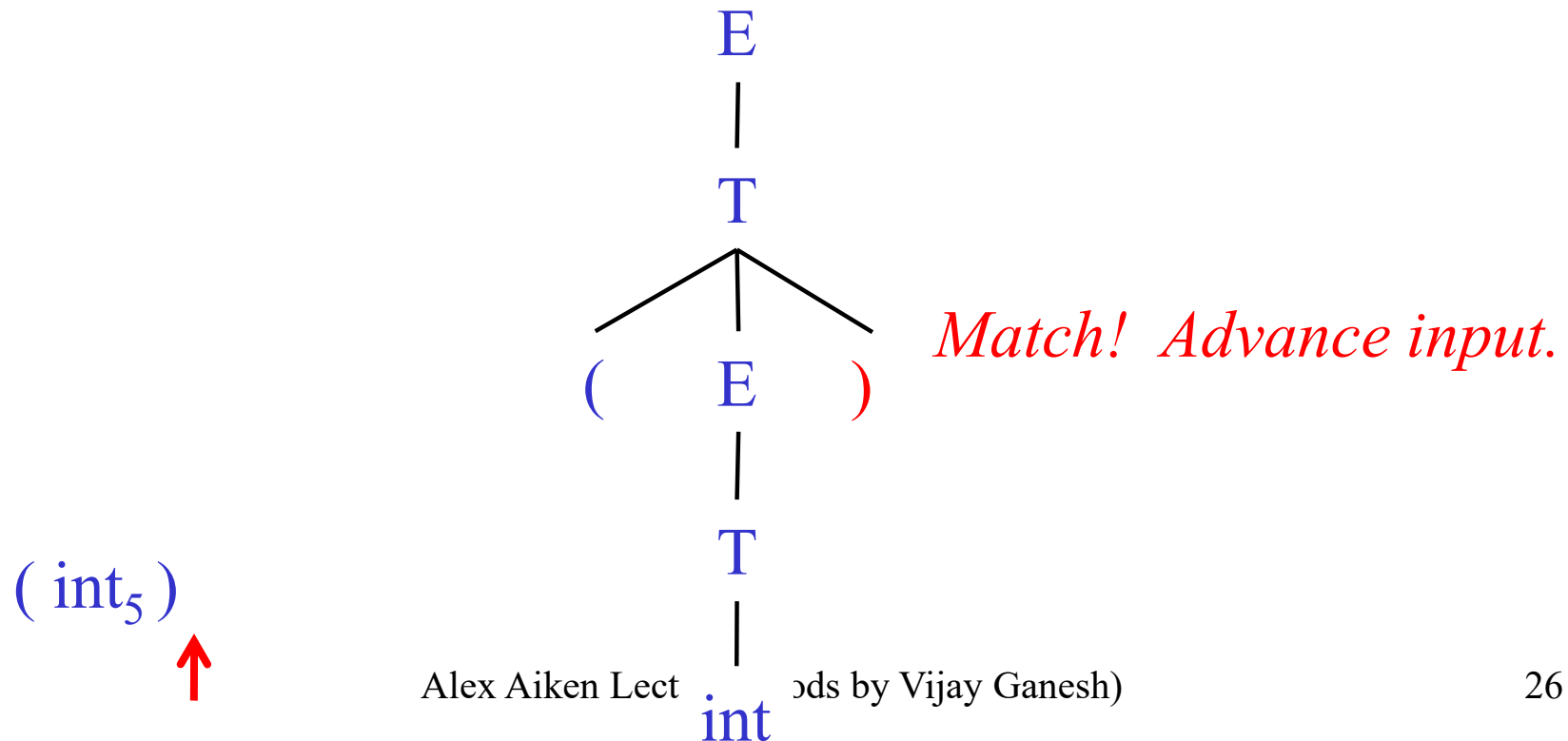
$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



Recursive Descent Parsing

$E \rightarrow T \mid T + E$

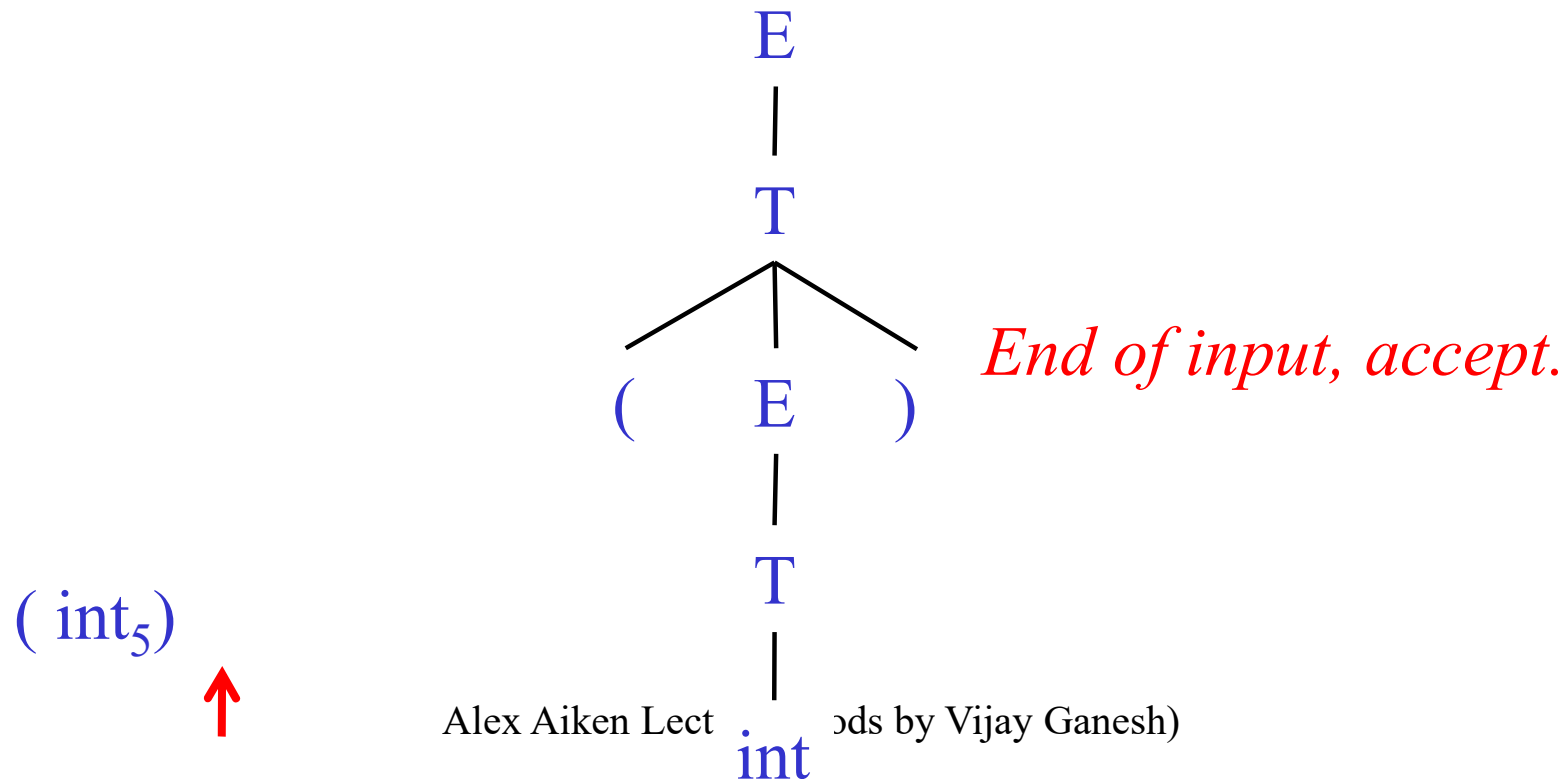
$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



A Recursive Descent Parser. Preliminaries

- Let TOKEN be the type of tokens
 - Special tokens INT, OPEN, CLOSE, PLUS, TIMES
- Let the global `next` point to the next token

A (Limited) Recursive Descent Parser (2)

- Define boolean functions that check the token string for a match of

- A given token terminal

```
bool term(TOKEN tok) { return *next++ == tok; }
```

- The nth production of S:

```
bool Sn() { ... }
```

- Try all productions of S:

```
bool S() { ... }
```

A (Limited) Recursive Descent Parser (3)

- For production $E \rightarrow T$
`bool E1() { return T(); }`
- For production $E \rightarrow T + E$
`bool E2() { return T() && term(PLUS) && E(); }`
- For all productions of E (with backtracking)
`bool E() {
 TOKEN *save = next;
 return (next = save, E1())
 || (next = save, E2()); }`

A (Limited) Recursive Descent Parser (4)

- Functions for non-terminal T

```
bool T1() { return term(INT); }
```

```
bool T2() { return term(INT) && term(TIMES) && T(); }
```

```
bool T3() { return term(OPEN) && E() && term(CLOSE); }
```

```
bool T() {  
    TOKEN *save = next;  
    return  (next = save, T1())  
           || (next = save, T2())  
           || (next = save, T3()); }
```

Recursive Descent Parsing. Notes.

- To start the parser
 - Initialize `next` to point to first token
 - Invoke `E()`
- Notice how this simulates the example parse
- Easy to implement by hand
 - But not completely general
 - Cannot backtrack once a production is successful
 - Works for grammars where at most one production can succeed for a non-terminal

Example

$E \rightarrow T \mid T + E$

(int)

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

```
bool term(TOKEN tok) { return *next++ == tok; }
```

```
bool E1() { return T(); }
```

```
bool E2() { return T() && term(PLUS) && E(); }
```

```
bool E() { TOKEN *save = next; return    (next = save, E1())  
                                           || (next = save, E2()); }
```

```
bool T1() { return term(INT); }
```

```
bool T2() { return term(INT) && term(TIMES) && T(); }
```

```
bool T3() { return term(OPEN) && E() && term(CLOSE); }
```

```
bool T() { TOKEN *save = next; return    (next = save, T1())  
                                           || (next = save, T2())  
                                           || (next = save, T3()); }
```

When Recursive Descent Does Not Work

- Consider a production $S \rightarrow S a$
 `bool S1() { return S() && term(a); }`
 `bool S() { return S1(); }`
- `S()` goes into an infinite loop
- A left-recursive grammar has a non-terminal S
 $S \rightarrow^+ S\alpha$ for some α
- Recursive descent does not work in such cases

Elimination of Left Recursion

- Consider the left-recursive grammar

$$S \rightarrow S \alpha \mid \beta$$

- S generates all strings starting with a β and followed by a number of α

- Can rewrite using right-recursion

$$S \rightarrow \beta S'$$

$$S' \rightarrow \alpha S' \mid \varepsilon$$

More Elimination of Left-Recursion

- In general

$$S \rightarrow S \alpha_1 \mid \dots \mid S \alpha_n \mid \beta_1 \mid \dots \mid \beta_m$$

- All strings derived from S start with one of β_1, \dots, β_m and continue with several instances of $\alpha_1, \dots, \alpha_n$
- Rewrite as

$$\begin{aligned} S &\rightarrow \beta_1 S' \mid \dots \mid \beta_m S' \\ S' &\rightarrow \alpha_1 S' \mid \dots \mid \alpha_n S' \mid \varepsilon \end{aligned}$$

General Left Recursion

- The grammar

$$S \rightarrow A \alpha \mid \delta$$

$$A \rightarrow S \beta$$

is also left-recursive because

$$S \rightarrow^+ S \beta \alpha$$

- This left-recursion can also be eliminated
- See Textbook or the Aho, Sethi and Ullman book for general algorithm

Summary of Recursive Descent

- Simple and general parsing strategy
 - Left-recursion must be eliminated first
 - ... but that can be done automatically
- Unpopular because of backtracking
 - Thought to be too inefficient
- In practice, backtracking is eliminated by restricting the grammar

Error Handling

- Purpose of the compiler is
 - To detect non-valid programs
 - To translate the valid ones
- Many kinds of possible errors (e.g. in C)

<u>Error kind</u>	<u>Example</u>	<u>Detected by ...</u>
Lexical	... \$...	Lexer
Syntax	... x *% ...	Parser
Semantic	... int x; y = x(3); ...	Type checker
Correctness	your favorite program	Tester/User

Syntax Error Handling

- Error handler should
 - Report errors accurately and clearly
 - Recover from an error quickly
 - Not slow down compilation of valid code
- Good error handling is not easy to achieve

Approaches to Syntax Error Recovery

- From simple to complex
 - Panic mode
 - Error productions
 - Automatic local or global correction
- Not all are supported by all parser generators

Error Recovery: Panic Mode

- Simplest, most popular method
- When an error is detected:
 - Discard tokens until one with a clear role is found
 - Continue from there
- Such tokens are called synchronizing tokens
 - Typically the statement or expression terminators

Syntax Error Recovery: Panic Mode (Cont.)

- Consider the erroneous expression

$(1 + + 2) + 3$

- Panic-mode recovery:
 - Skip ahead to next integer and then continue
- Bison: use the special terminal **error** to describe how much input to skip

$E \rightarrow \text{int} \mid E + E \mid (E) \mid \text{error int} \mid (\text{error})$

Syntax Error Recovery: Error Productions

- Idea: specify in the grammar known common mistakes
- Essentially promotes common errors to alternative syntax
- Example:
 - Write **5 x** instead of **5 * x**
 - Add the production $E \rightarrow \dots \mid E E$
- Disadvantage
 - Complicates the grammar

Error Recovery: Local and Global Correction

- Idea: find a correct “nearby” program
 - Try token insertions and deletions
 - Exhaustive search
- Disadvantages:
 - Hard to implement
 - Slows down parsing of correct programs
 - “Nearby” is not necessarily “the intended” program
 - Not all tools support it

Syntax Error Recovery: Past and Present

- Past
 - Slow recompilation cycle (even once a day)
 - Find as many errors in one cycle as possible
 - Researchers could not let go of the topic
- Present
 - Quick recompilation cycle
 - Users tend to correct one error/cycle
 - Complex error recovery is less compelling
 - Panic-mode seems enough