PARSING

Baishakhi Ray

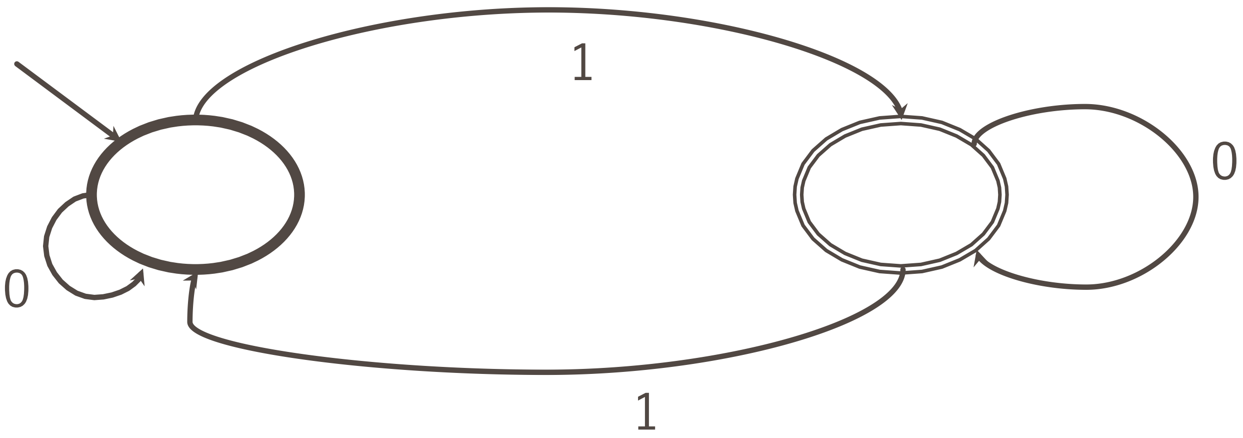
Fall 2018

Intro to Parsing

- Regular Languages
  - Weakest formal languages that are widely used
  - Many applications
- Consider the language $\{(^i\ )^i \mid i \geq 0\}$
  - (), (( )), ((( )))
  - ((1 + 2) * 3)
- Nesting structures
  - if .. if.. else.. else..

Regular languages cannot handle well

Automata that accepts odd numbers of 1



How many 1s it has accepted?

- Only solution is duplicate state

Automata does not have any memory

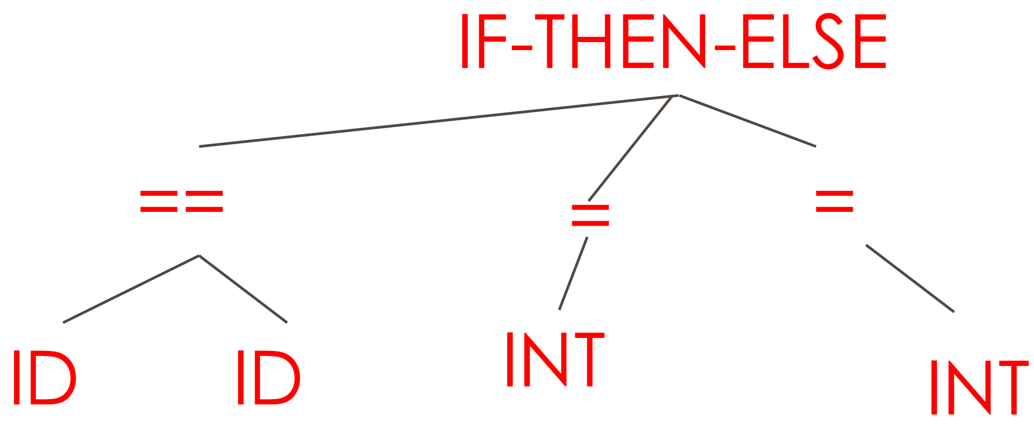
Intro to Parsing

- Input: if(x==y) 1 else 2;

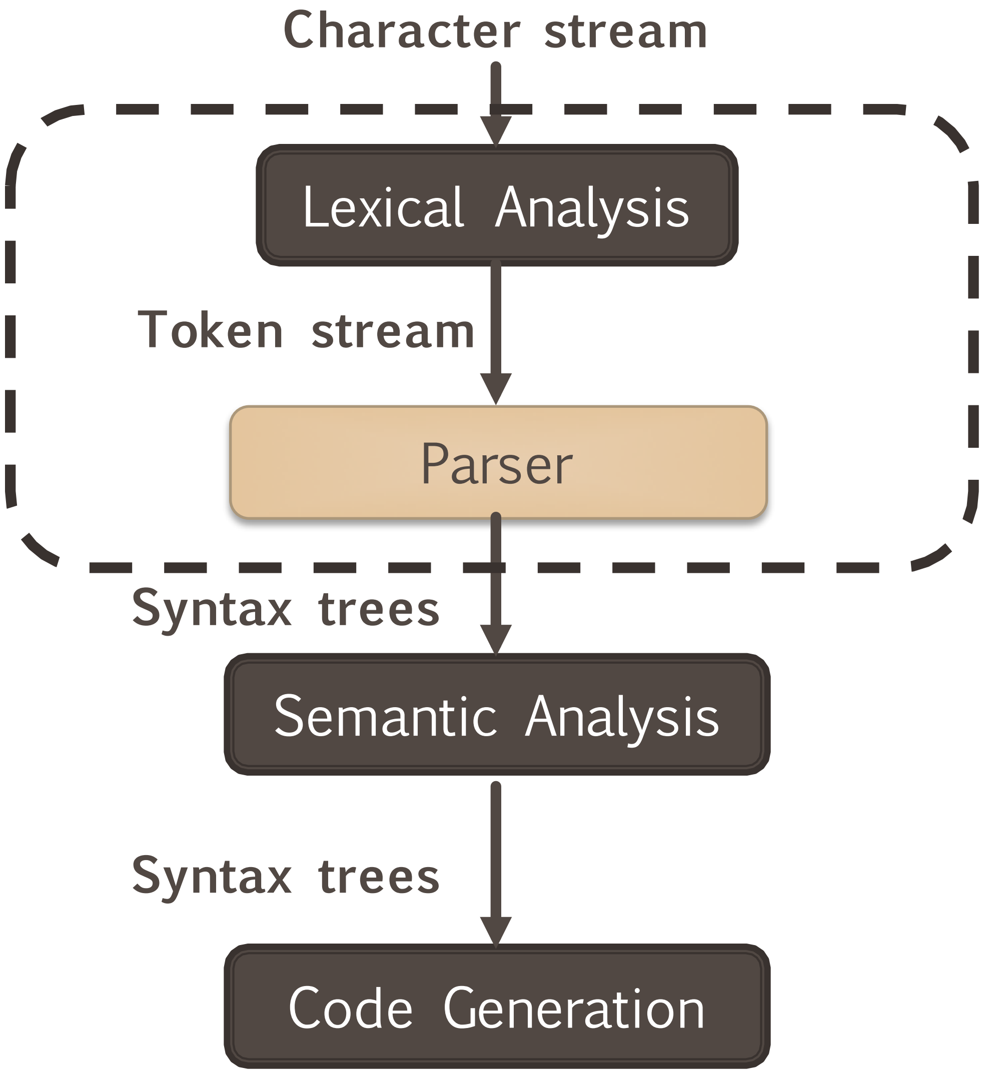
- Parser Input (Lexical Input):

  KEY(IF) ‘(‘ ID(x) OP(‘==‘) ‘)’ INT(1) KEY(ELSE) INT(2) ‘;’

- Parser Output



Intro to Parsing



- Nor every strings of tokens are valid
- Parser must distinguish between valid and invalid token strings.
- We need
  - A Language: to describe valid string
  - A method: to distinguish valid from invalid.

Context Free Grammar

- A CFG consists of
  - A set of terminal T
  - A set of non-terminal N
  - A start symbol S (S ϵ N)
  - A set of production rules
    - X -> $Y_1 \ldots\ldots Y_N$
    - X ϵ N
    - Y_i ϵ {N, T, ε}

- Ex: S -> (S) | ε
  - N = {S}
  - T = { (,) , ε}

Context Free Grammar

1. Begin with a string with only the start symbol S
2. Replace a non-terminal X with in the string by the RHS of some production rule: $X \text{->} Y_1 \ldots\ldots Y_n$
3. Repeat 2 again and again until there are no non-terminals

$X_1 \ldots\ldots X_i \, \underline{X} \, X_{i+1} \; \ldots. \; X_n \;\text{->}\; X_1 \ldots\ldots X_i \, Y_1 \ldots\ldots Y_k \, X_{i+1} \; \ldots. \; X_n$

For the production rule $X \text{->} Y_1 \ldots\ldots Y_k$

$$\propto_0 \rightarrow \propto_1 \rightarrow \ \ldots \rightarrow \propto_n$$

$$\propto_0 \xrightarrow{*} \propto_n, \; n \geq 0$$

Context Free Grammar

- Let G be a CFG with start symbol S. Then the language L(G) of G is:

$$\{a_1 \ldots \ldots ..an | \forall_i\, ai \ \in T \ \wedge S \overset{*}{\rightarrow} a_1 a_2 \ldots \ldots an\}$$

Context Free Grammar

- There are no rules to replace terminals.
- Once generated, terminals are permanent
- Terminals ought to be tokens of programming languages
- Context-free grammars are a natural notation for this recursive structure

CFG: Simple Arithmetic expression

E → E + E

| E * E

| (E)

| id

Languages can be generated: id, (id), (id + id) * id, …

Derivation

- A derivation is a sequence of production
  - S -> … -> … ->

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

- Grammar
  - E -> E + E | E * E | (E) | id
- String
  - id * id + id
- Derivation

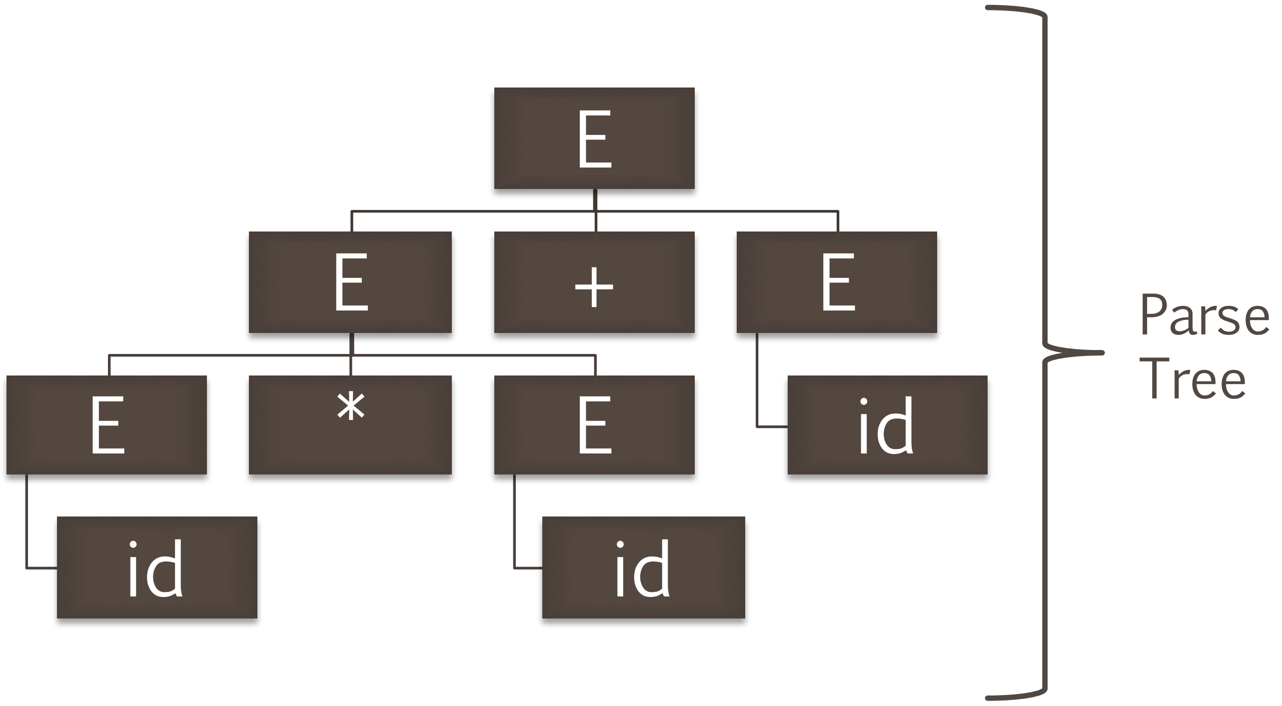
E -> E + E

-> E * E + E

-> id * E + E

-> id * id + E

-> id * id + id



Parse Tree

- 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

Parse Tree

- Left-most derivation
  - At each step, replace the left-most non-terminal

E -> E + E

-> E * E + E

-> id * E + E

-> id * id + E

-> id * id + id

- Right-most derivation
  - At each step, replace the right-most non-terminal

E -> E + E

-> E + id

-> E * E + id

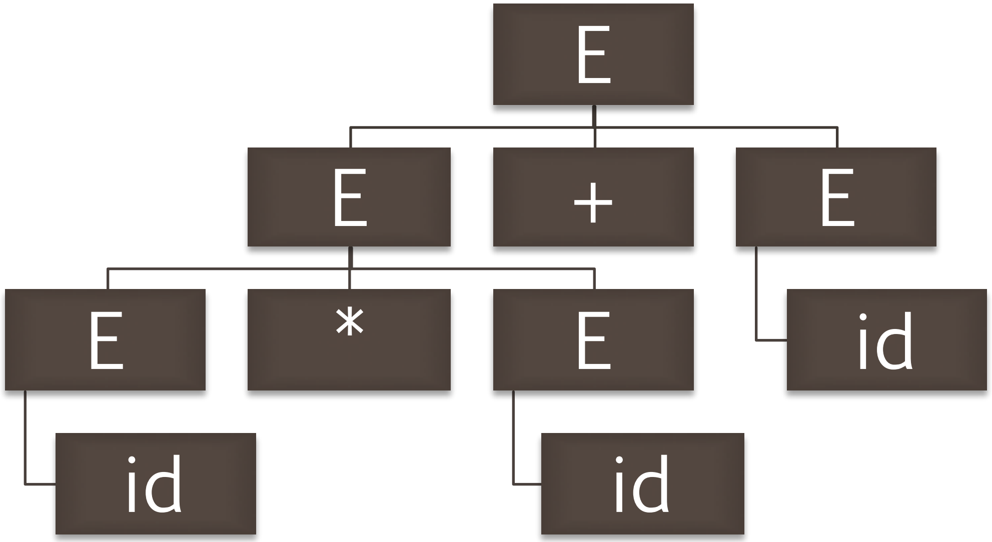
-> E * id + id

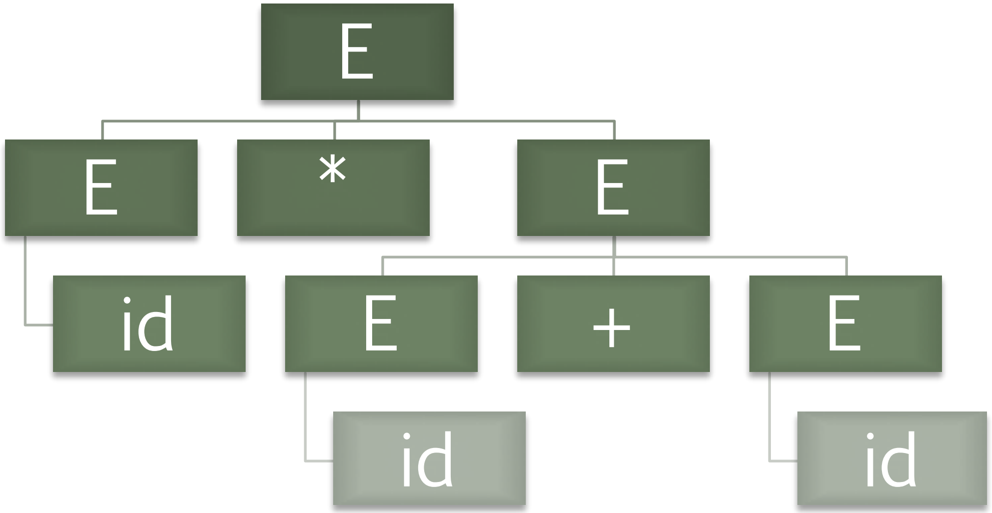
-> id * id + id

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

Ambiguity

- Grammar
  - E -> E + E | E * E | (E) | id
- String
  - id * id + id





Ambiguity

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

- Ambiguity is bad
  - Leaves meaning for some programs ill-defined

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)

Error Kind	Example	Detected by
Lexical	... $ ...	Lexer
Syntax	... x*%...	Parser
Semantic	... int x; y = x(3);...	Type Checker
Correctness	your program	tester/user

Error Handling

- Error Handler should
  - Recover errors accurately and quickly
  - Recover from an error quickly
  - Not slow down compilation of valid code

- Types of Error Handling
  - Panic mode
  - Error productions
  - Automatic local or global correction

Panic Mode Error Handling

- Panic mode is simplest and most popular method

- When an error is detected
  - Discard tokens until one with a clear role is found
  - Continue from there

- Typically looks for “synchronizing” tokens
  - Typically the statement of expression terminators

Panic Mode Error Handling

- Example:
  - (1 + + 2) + 3
- Panic-mode recovery:
  - Skip ahead to the next integer and then continue
- Bison: use the special terminal error to describe how much input to skip
  - $E \to \underbrace{\text{int} \mid E + E \mid (E)}_{\text{Normal mode}} \mid \underbrace{\text{error int} \mid (\text{error})}_{\text{Error mode}}$

Error Productions

- Specify known common mistakes in the grammar

- Example:
  - Write 5x instead of 5 * x
  - Add production rule E -> .. | E E

- Disadvantages
  - complicates the grammar

Error Corrections

- Idea: find a correct “nearby” program
  - Try token insertions and deletions (goal: minimize edit distance)
  - Exhaustive search

- Disadvantages
  - Hard to implement
  - Slows down parsing of correct programs
  - “Nearby” is not necessarily “the intended” program

Error Corrections

- Past
  - Slow recompilation cycle (even once a day)
  - Find as many errors in once cycle as possible

- Disadvantages
  - Quick recompilation cycle
  - Users tend to correct one error/cycle
  - Complex error recovery is less compelling

Abstract Syntax Trees

- A parser traces the derivation of a sequence of tokens

- But the rest of the compiler needs a structural representation of the program

- Abstract Syntax Trees
  - Like parse trees but ignore some details
  - Abbreviated as AST

Abstract Syntax Trees

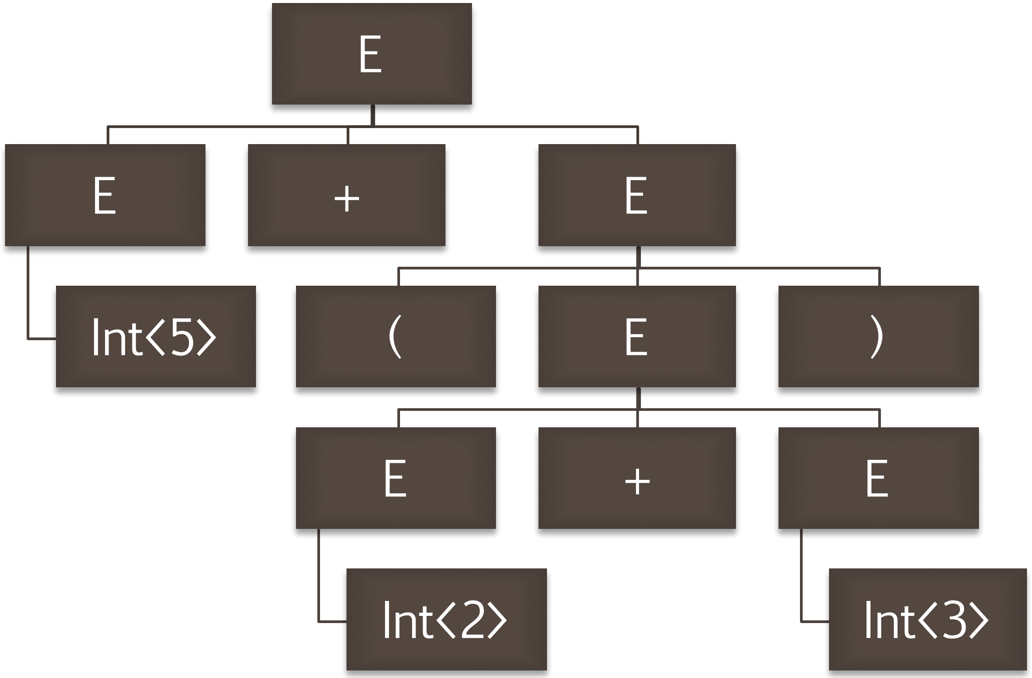
- Grammar
  - E -> int | (E) | E + E

- String
  - 5 + (2 + 3)

- After lexical analysis
  - Int<5> ‘+’ ‘(‘ Int<2> ‘+’ Int<3> ‘)’

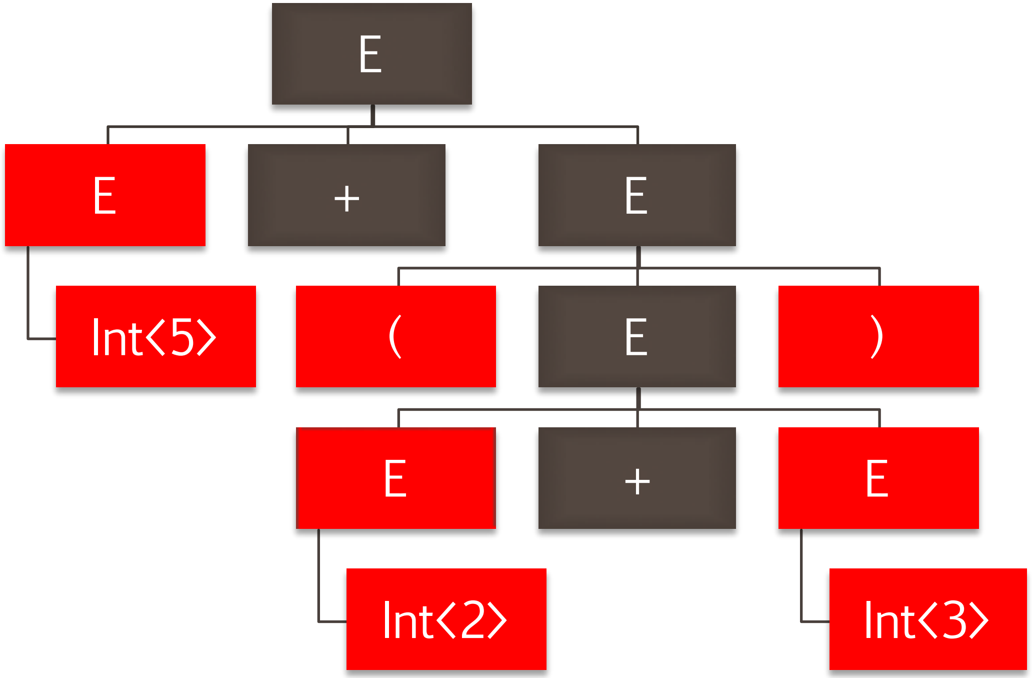
Abstract Syntax Trees: 5 + (2 + 3)

Parse Trees



Abstract Syntax Trees: 5 + (2 + 3)

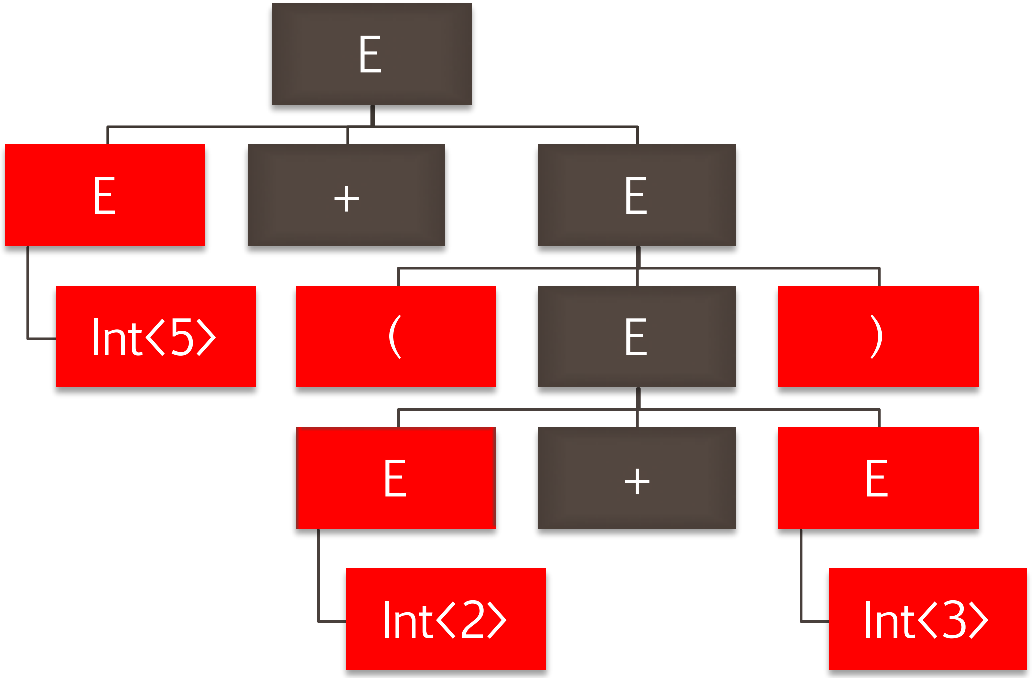
Parse Trees



- Have too much information
  - Parentheses
  - Single-successor nodes

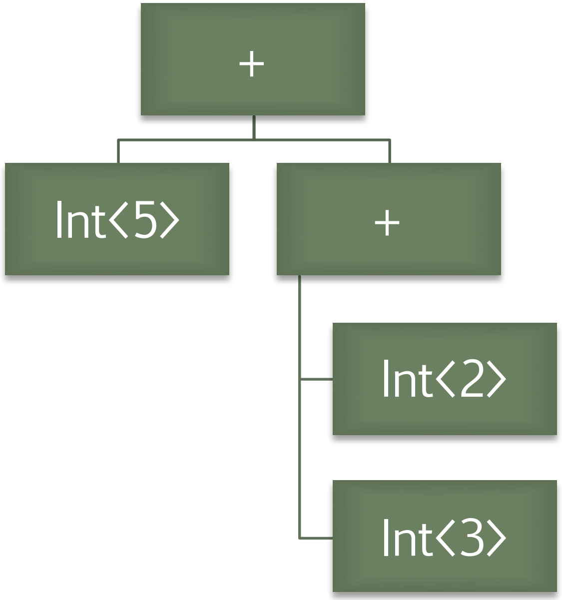
Abstract Syntax Trees: 5 + (2 + 3)

Parse Trees



- Have too much information
  - Parentheses
  - Single-successor nodes

AST



- ASTs capture the nesting structure
- But abstracts from the concrete syntax
  - More compact and easier to use

Parsing algorithm: Recursive Descent Parsing

- The parse tree is constructed
  - From the top
  - From left to right

- Terminals are seen in order of appearance in the token stream

Parsing algorithm: Recursive Descent Parsing

- Grammar:
  - E -> T | T + E
  - T -> int | int * T | (E)

- Token Stream: (int<5>)

- Start with top level non-terminal E
  - Try the rules for E in order

Recursive Descent Parsing Example

E -> **T** | T + E

T -> **int** | int * T | (E)

E

T

int

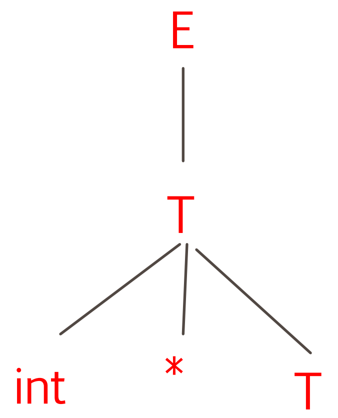
mismatch: int does not match arrowhead (
backtrack

(int<5>)

Recursive Descent Parsing Example

E -> **T** | T + E

T -> int | **int * T** | (E)



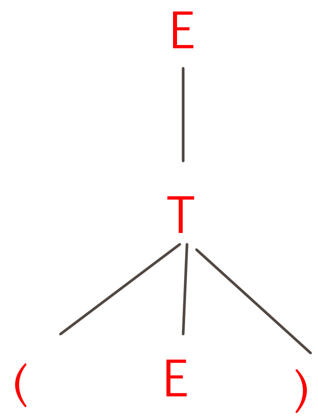
backtrack

(int<5>)

Recursive Descent Parsing Example

E -> **T** | T + E

T -> int | int * T | **(E)**



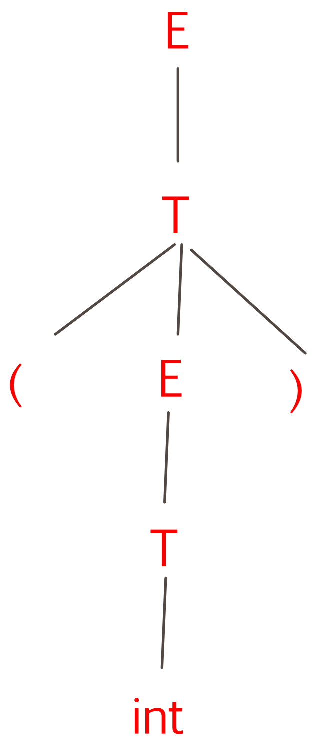
Match! Advance input

(int<5>)

Recursive Descent Parsing Example

E -> **T** | T + E

T -> **int** | int * T | (E)



Match! Advance input

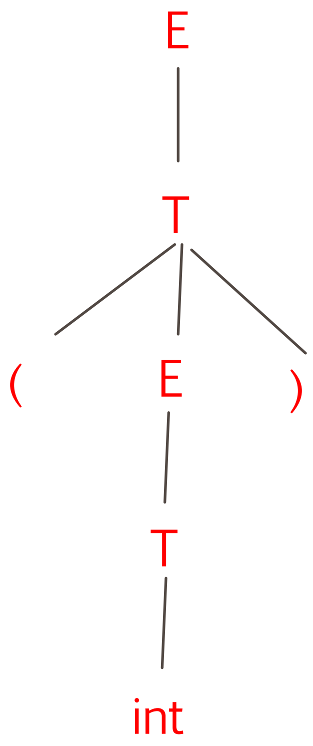
(int<5>)

↑

Recursive Descent Parsing Example

E -> T | T + E

T -> int | int * T | (E)



Match! Advance input

(int<5>)