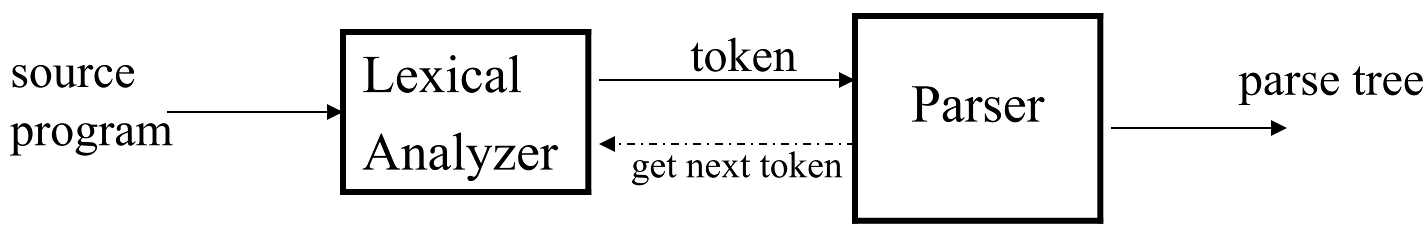
Parser

- Parser works on a stream of tokens.

- The smallest item is a token.



Syntax Analysis

- Context Free Grammar (CFG)
- Derivation
- Concrete and Abstract syntax tree
- Ambiguity

Formal Language

- An **Alphabet** is a set Σ of input symbols, that act as letters
- A **Language** over Σ is a set of strings made from symbols in Σ
- When scanning, our alphabets are ASCII and we produced tokens
- When parsing, our alphabets are set of tokens produced from Scanner.

The limit of Regular Language

- Regular Expression are weak to define programing languages

- Cannot define a regular expression matching all functions with properly nested block structure.

- We need more powerful formalism-(CFG) Context Free Grammar which is a superset of RE.

- The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.

A context-free grammar

- gives a precise syntactic specification of a programming language.
- the design of the grammar is an initial phase of the design of a compiler.
- a grammar can be directly converted into a parser by some tools.

Formal Definition of CFG

- CFG is a collection of 4 objects:
  - Set of non terminals / variables
  - Set of terminals
  - Set of production rules
  - Start symbol

Non Terminals: E, Op

*Terminals: int, (,), +, -, *, /*

$E \rightarrow \texttt{int}$

$E \rightarrow E\ Op\ E$

$E \rightarrow \texttt{(}E\texttt{)}$

$Op \rightarrow \texttt{+}$

$Op \rightarrow \texttt{-}$

$Op \rightarrow \texttt{*}$

$Op \rightarrow \texttt{/}$

Arithmetic Expression

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:
- Input String is: *int * (int + int)*

E → int

E → E Op E

E → (E)

Op → +

Op → -

Op → *

Op → /

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E * (E Op E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int Op int)

⇒ int * (int + int)

CFG for Programing 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: sequence of steps to derive a string

$$
\begin{aligned}
& \mathbf{E} \\
\Rightarrow\ & \mathbf{E\ Op\ E} \\
\Rightarrow\ & \mathbf{E\ Op\ (E)} \\
\Rightarrow\ & \mathbf{E\ Op\ (E\ Op\ E)} \\
\Rightarrow\ & \mathbf{E} * \mathbf{(E\ Op\ E)} \\
\Rightarrow\ & \mathtt{int} * \mathbf{(E\ Op\ E)} \\
\Rightarrow\ & \mathtt{int} * (\mathtt{int}\ \mathbf{Op\ E}) \\
\Rightarrow\ & \mathtt{int} * (\mathtt{int}\ \mathbf{Op}\ \mathtt{int}) \\
\Rightarrow\ & \mathtt{int} * (\mathtt{int} + \mathtt{int})
\end{aligned}
$$

Leftmost Derivations and Right most 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.

Leftmost Derivation

$$
\begin{aligned}
\text{BLOCK} &\rightarrow \text{STMT} \\
&\mid\ \texttt{\{}\ \text{STMTS}\ \texttt{\}} \\[1em]
\text{STMTS} &\rightarrow \varepsilon \\
&\mid\ \text{STMT STMTS} \\[1em]
\text{STMT} &\rightarrow \text{EXPR}\texttt{;} \\
&\mid\ \texttt{if (}\text{EXPR}\texttt{)}\ \text{BLOCK} \\
&\mid\ \texttt{while (}\text{EXPR}\texttt{)}\ \text{BLOCK} \\
&\mid\ \texttt{do}\ \text{BLOCK}\ \texttt{while (}\text{EXPR}\texttt{);} \\
&\mid\ \text{BLOCK} \\
&\mid\ \ldots \\[1em]
\text{EXPR} &\rightarrow \texttt{identifier} \\
&\mid\ \texttt{constant} \\
&\mid\ \text{EXPR} + \text{EXPR} \\
&\mid\ \text{EXPR} - \text{EXPR} \\
&\mid\ \text{EXPR} * \text{EXPR} \\
&\mid\ \text{EXPR} = \text{EXPR}
\end{aligned}
$$

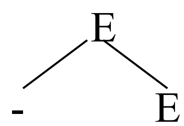
$$
\begin{aligned}
& \text{STMTS} \\
\Rightarrow\ & \text{STMT STMTS} \\
\Rightarrow\ & \text{EXPR}\texttt{;}\ \text{STMTS} \\
\Rightarrow\ & \text{EXPR} \texttt{ = } \text{EXPR}\texttt{;}\ \text{STMTS} \\
\Rightarrow\ & \texttt{id = } \text{EXPR}\texttt{;}\ \text{STMTS} \\
\Rightarrow\ & \texttt{id = } \text{EXPR} \texttt{ + } \text{EXPR}\texttt{;}\ \text{STMTS} \\
\Rightarrow\ & \texttt{id = id + } \text{EXPR}\texttt{;}\ \text{STMTS} \\
\Rightarrow\ & \texttt{id = id + constant;}\ \text{STMTS} \\
\Rightarrow\ & \texttt{id = id + constant;}
\end{aligned}
$$

Parse Tree

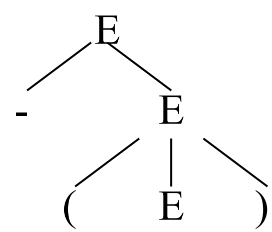
- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.

- A parse tree can be seen as a graphical representation of a derivation.

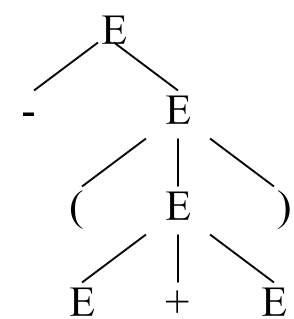
$E \Rightarrow -E$



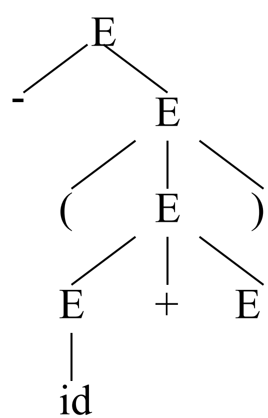
$\Rightarrow -(E)$



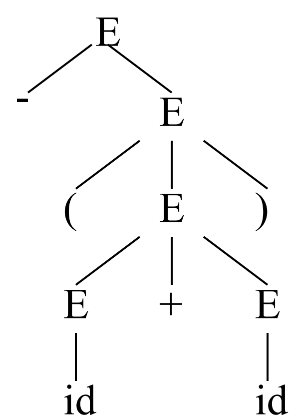
$\Rightarrow -(E+E)$



$\Rightarrow -(id+E)$



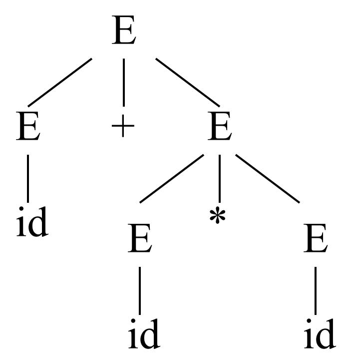
$\Rightarrow -(id+id)$



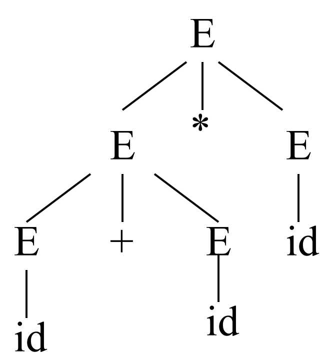
Ambiguity

- A grammar produces more than one parse tree for a sentence is called as an ***ambiguous*** grammar.

$$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E$$
$$\Rightarrow id+id*E \Rightarrow id+id*id$$



$$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E$$
$$\Rightarrow id+id*E \Rightarrow id+id*id$$



Ambiguity (cont.)

- For the most parsers, the grammar must be unambiguous.

- unambiguous grammar

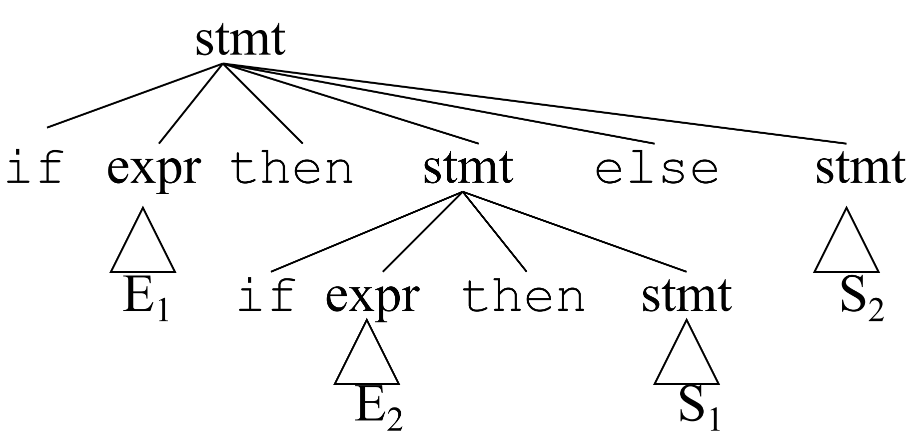
  ➔ unique selection of the parse tree for a sentence

- We should eliminate the ambiguity in the grammar during the design phase of the compiler.
- An unambiguous grammar should be written to eliminate the ambiguity.
- We have to prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice.

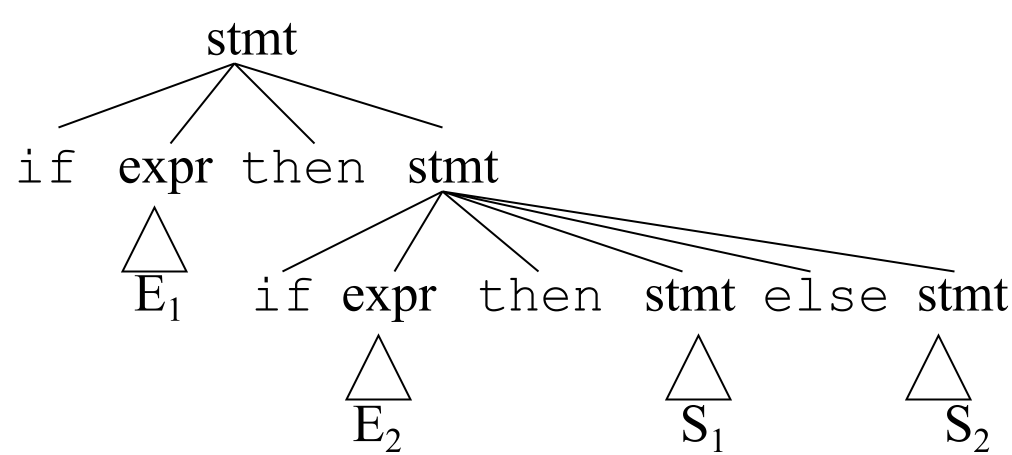
Ambiguity (cont.)

stmt → `if` expr `then` stmt |
      `if` expr `then` stmt `else` stmt | otherstmts

`if` E_1 `then` `if` E_2 `then` S_1 `else` S_2



1



2

Ambiguity (cont.)

- We prefer the second parse tree (else matches with closest if).
- So, we have to disambiguate our grammar to reflect this choice.

- The unambiguous grammar will be:

stmt → matchedstmt | unmatchedstmt

matchedstmt → `if` expr `then` matchedstmt `else` matchedstmt | otherstmts

unmatchedstmt → `if` expr `then` stmt |
`if` expr `then` matchedstmt `else` unmatchedstmt

The ambiguity

$R \to$ a | b | c | ...

$R \to$ "ε"

$R \to RR$

$R \to R$ "|" R

$R \to R*$

$R \to$ (R)

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.

Allows for unambiguous parsing of ambiguous grammars

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.

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

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.

Parsers

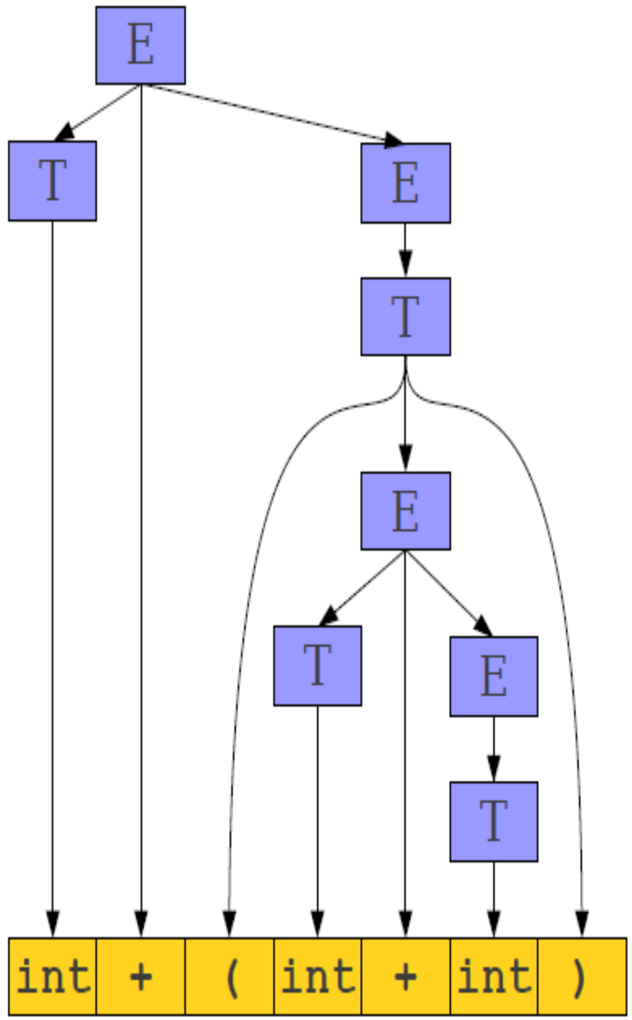
- We categorize the parsers into two groups:

1. **Top-Down Parser**
   - the parse tree is created top to bottom, starting from the root.
   - Beginning with the start symbol, try to guess the productions to apply to end up at the user's program

2. **Bottom-Up Parser**
   - the parse is created bottom to top; starting from the leaves
   - Beginning with the user's program, try to apply productions in reverse to convert the program back into the start symbol

Top Down Parser

$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow int$
$T \rightarrow (E)$



$E \Rightarrow T + E$
$E \Rightarrow int + T$
$E \Rightarrow int + (E)$
$E \Rightarrow int + (T + E)$
$E \Rightarrow int + (int + T)$
$E \Rightarrow int + (int + int)$

Challenges in Top-Down Parsing

- Top-down parsing begins with virtually no information.
  - Begins with just the start symbol, which matches e*very* program
- How can we know which productions to apply?
- In general, we can't.
  - There are some grammars for which the best we can do is guess and backtrack if we're wrong.
  - If we have to guess, how do we do it?

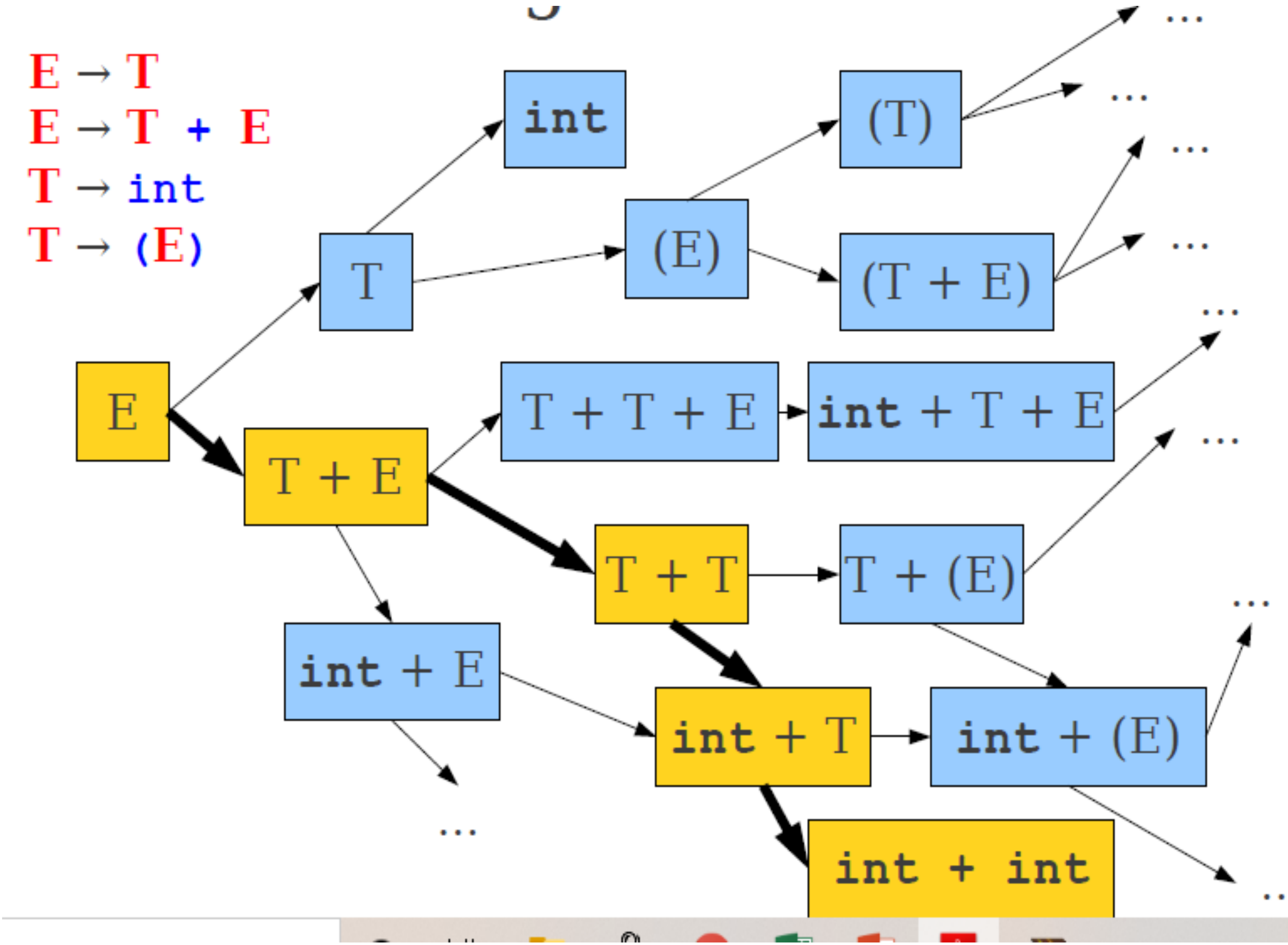
Parsing as a Search

- An idea: **treat parsing as a graph search**.

- Each node is a **sentential form** (a string of terminals and nonterminals derivable from the start symbol).

- There is an edge from node $\boldsymbol{\alpha}$ to node $\boldsymbol{\beta}$ iff $\boldsymbol{\alpha} \Rightarrow \boldsymbol{\beta}$.

Parsing as a search

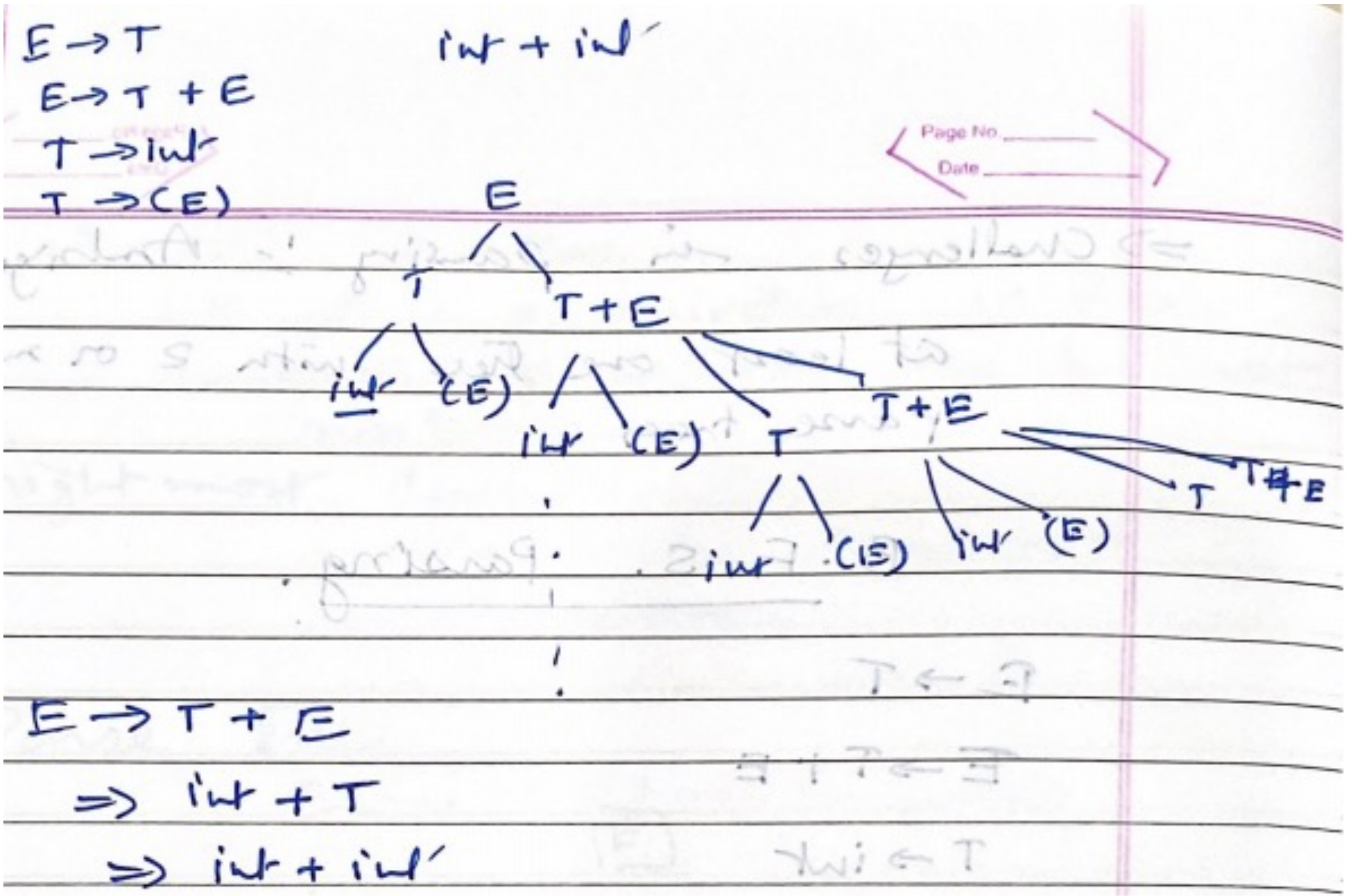


Top Down Approach

Breadth-First Search

- Maintain a worklist of sentential forms, initially just the start symbol **S**.
- While the worklist isn't empty:
  - Remove an element from the worklist.
  - If it matches the target string, you're done.
  - Otherwise, for each possible string that can be derived in one step, add that string to the worklist.
- Can recover a parse tree by tracking what productions we applied at each step.

BFS Example:


E → T
E → T + E
T → int
T → (E)
int + int
E
T
T + E
int
(E)
int
(E)
T
T + E
int
(E)
int
(E)
T
T + E
E → T + E
⇒ int + T
⇒ int + int

BFS is Slow

Enormous time and memory usage:

- Lots of **wasted effort**:

  Generates a lot of sentential forms that couldn't possibly match.

  But in general, extremely hard to tell whether a sentential form can match – that's the job of parsing!

- High **branching factor**:

  Each sentential form can expand in (potentially) many ways for each nonterminal it contains.

Reducing Wasted Effort

- Suppose we're trying to match a string X.
  - Suppose we have a sentential form T = aB, where a is a string of terminals and B is a string of terminals and non terminals.
  - If a isn't a prefix of X, then no string derived from T can ever match X.
  - If we can find a way to try to get a prefix of terminals at the front of our sentential forms, then we can start pruning out impossible options.

Reducing the Branching Factor

- If a string has many non terminals in it, the branching factor can be high.
  - Sum of the number of productions of each nonterminal involved.

- If we can restrict which productions we apply, we can keep the branching factor lower.

Leftmost Derivations

- Recall: A **leftmost derivation** is one where we always expand the leftmost symbol first.
- Updated algorithm:
  - Do a breadth-first search, **only considering leftmost derivations**.
    - Drops branching factor.
    - Increases likelihood that we get a prefix of non terminals.
  - Prune sentential forms that can't possibly match.
    - Avoids wasted effort

Leftmost derivation example:

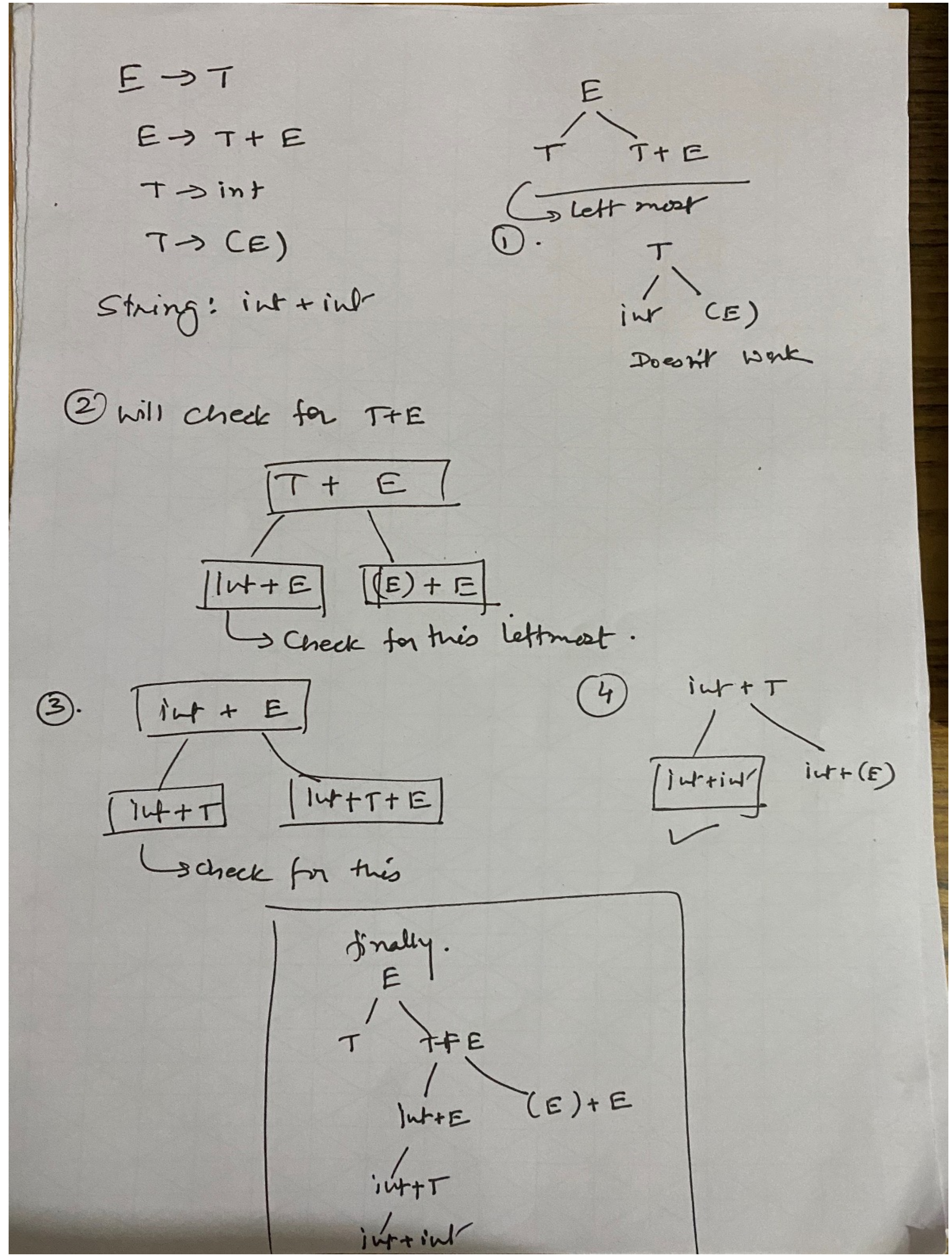
E → T

E → T + E

T → int

T → (E)

String: int + int



Leftmost BFS

- Substantial improvement over previous method.
- Will always find a valid parse of a program if one exists.
- Can easily be modified to find if a program can't be parsed.
- But, there are still problems. Grammar like
  - **A** → **Aa** | **Ab** | **c** creates exponential time and memory for the string: caaaaaaaaa
  - Go for another approach

Leftmost DFS

- Idea: Use **depth-first** search.
  - Advantages:
  - Lower memory usage: Only considers one branch at a time.
  - High performance: On many grammars, runs very quickly.
- Easy to implement: Can be written as a set of mutually recursive functions.

Example:

Left most DFS.

2

$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow int$

$T \rightarrow (E)$

1.

$$E \rightarrow T \rightarrow int$$

$\rightarrow$ Not possible.

2

$$E \rightarrow T \rightarrow (E)$$

– not possible

3

$$E \rightarrow T + E \rightarrow int + E \rightarrow int + T \rightarrow int + int$$

Problems with Leftmost DFS

$A \rightarrow Aa \mid c$

A
Aa
Aaa
Aaaa
Aaaaa

- A non Terminal A is said to be left recursive iff A ➔ Aw for some string w.
- Leftmost DFS may fail in left recursive grammar.
- It is possible to eliminate left recursion

Summary of Leftmost BFS / DFS

BFS	DFS
Leftmost BFS works on all grammars	Leftmost DFS works on grammars without left recursion.
Worst-case runtime is exponential.	Worst-case runtime is exponential.
Worst-case memory usage is exponential.	Worst-case memory usage is linear.
Rarely used in practice.	Often used in a limited form as **recursive descent parser**.

Tradeoffs in Prediction

- Predictive parsers are *fast*.
  - Many predictive algorithms can be made to run in linear time.
  - Often can be table-driven for extra performance.
- Predictive parsers are *weak*.
  - Not all grammars can be accepted by predictive parsers.
- Trade *expressiveness* for *speed*.

Lookahead Symbols

- Given just the start symbol, how do you know which productions to use to get to the input program?

- Idea: Use **lookahead tokens**.

- When trying to decide which production to use, look at some number of tokens of the input to help make the decision

Predictive Parsing

- The leftmost DFS/BFS algorithms are **backtracking** algorithms.
  - Guess which production to use, then back up if it doesn't work.
- There is another class of parsing algorithms called **predictive** algorithms.
  - Based on remaining input, predict (*without backtracking*) which production to use

Implementing Predictive Parsing

- Predictive parsing is only possible if we can predict which production to use given some number of lookahead tokens.
- Increasing the number of lookahead tokens, increases the number of grammars we can parse, but complicates the parser.
- Decreasing the number of lookahead tokens, decreases the number of grammars we can parse, but simplifies the parser.

Predictive Parsing

E → T
E → T + E
T → int
T → (E)

E
T + E
int + E
int + T
int + (E)
int + (T + E)
int + (int + E)
int + (int + T)
int + (int + int)

int	+	(	int	+	int	)

Code for Recursive Descent Parser (RDP)

$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

Procedure E()

: Call Procedure T()

| Call Procedure T(), match "+" , Call Procedure E()

Procedure T()

: match "int"

| match "("

Call Procedure E,

match ")"