Syntax Analysis
Comparison with Lexical Analysis

The second phase of compilation

<table>
<thead>
<tr>
<th>Phase</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexer</td>
<td>string of characters</td>
<td>string of tokens</td>
</tr>
<tr>
<td>Parser</td>
<td>string of tokens</td>
<td>Parse tree/AST</td>
</tr>
</tbody>
</table>
What Parse Tree?

- A **parse tree** represents the program structure of the input.
- Programming language constructs usually have recursive structures.

\[
\text{If-stmt} \equiv \text{if (EXPR) then Stmt else Stmt fi}
\]

\[
\text{Stmt} \equiv \text{If-stmt} \mid \text{While-stmt} \mid \ldots
\]
A Parse Tree Example

- Code to be compiled:
  
  \[ ... \text{if } x==y \text{ then } \ldots \text{ else } \ldots \text{ fi } \]

- Lexer:  

- Parser:
  
  - Input: sequence of tokens
    
    \[ ... \text{IF } \text{ID}==\text{ID} \text{ THEN } \ldots \text{ELSE } \ldots \text{ FI } \]
  
  - Desired output:

    \[
    \text{IF-STMT}
    \]

    \[
    \begin{array}{c}
    == \\
    \text{STMT} \\
    \text{STMT} \\
    \end{array}
    \]

    \[
    \begin{array}{c}
    \text{ID} \\
    \text{ID} \\
    \end{array}
    \]
What Formalism to Use?

- How to represent the program structure?
  - Is it possible to use RE/FA?
    RE(Regular Expression) \equiv FA(Finite Automata)
What Formalism to Use?

- How to represent the program structure?
  - Is it possible to use RE/FA?
    RE(Regular Expression) \equiv FA(Finite Automata)

- RE/FA is not powerful enough

Example: matching parenthesis: \# of “(” equals \# of “)”

- \( (x+y)^*z \)
- \( ((x+y)+y)^*z \)
- \( \ldots \)
- \( \ldots(((x+y)+y)+y)\ldots \)
- \( ((x+y)+y)+y)^*z \)
- \( ((x+y)+y)+y)^*z \)
RE/FA is Not Powerful Enough

- Describe strings with pattern $[\[\]'i (i \geq 1)$
RE/FA is Not Powerful Enough

- Describe strings with pattern \([i]^{i} \ (i \geq 1)\)
  - “[”, “]” should be in different states
  - “[”, “[[” should be in different states

```
\[
\begin{align*}
&\text{[} \quad \text{]} \quad \text{[}] \\
\text{[} \quad \text{]} \quad \text{[}] \\
\text{[[} \quad \text{]} \quad \text{[]]}
\end{align*}
\]
```
RE/FA is Not Powerful Enough

- Describe strings with pattern $[i]^j$ ($i \geq 1$)
  - "[", "[]" should be in different states
  - "[", "[[" should be in different states

```
[ ]  \[ \]
[ ] x

[[ ] [ ]]
```

- "[[[..[" should be in a new state
- Since $i$ can be any positive integer value, the number of states is infinite
- Contradiction: FA — finite automata
Formalism for Syntax Analysis

- We need a more powerful formalism for describing language constructs
  - CFL (context free language) concept

- Before discussing CFL, let us generalize language definition
  - Covers both RE and CFL
  - and more ...
Recall language definition
- Language — set of strings over alphabet
  - Alphabet: finite set of symbols
  - Null string: $\epsilon$
  - Sentences: strings in the language

It is possible to describe a language using a grammar
- Like define English using English grammars
An Example

Language \( L = \{ \text{any string with “00” at the end} \} \)

Grammar \( G = \{ T, N, s, \delta \} \)

where \( T = \{ 0, 1 \}, N = \{ A, B \}, s = A, \) and

grammar rule set \( \delta = \{ A \rightarrow 0A \mid 1A \mid 0B, B \rightarrow 0 \} \)

Derivation: from grammar to language

- \( A \Rightarrow 0A \Rightarrow 00B \Rightarrow 000 \)
- \( A \Rightarrow 1A \Rightarrow 10B \Rightarrow 100 \)
- \( A \Rightarrow 0A \Rightarrow 00A \Rightarrow 000B \Rightarrow 0000 \)
- \( A \Rightarrow 0A \Rightarrow 01A \Rightarrow ... \)
A grammar consists of 4 components \((T, N, s, \delta)\)

- **T** — set of terminal symbols
  - Essentially tokens — leaves in the parse tree

- **N** — set of non-terminal symbols
  - Internal nodes in the parse tree
  - One or more symbols grouped into higher construct
    example: declaration, statement, loop, ...

- **s** — the non-terminal start symbol where derivation starts

- **\(\delta\)** — a set of production rules
  - “LHS \(\rightarrow\) RHS”: left-hand-side produces right-hand-side
Production Rule and Derivation

“LHS $\rightarrow$ RHS”
- to replace LHS with RHS
- it specifies how to transform one string to another

$\beta \Rightarrow \alpha$: string $\beta$ derives $\alpha$
- $\beta \Rightarrow \alpha$ — 1 step
- $\beta \Rightarrow \ast \alpha$ — 0 or more steps
- $\beta \Longrightarrow \alpha$ — 0 or more steps

example:
- $A \Rightarrow 0A \Rightarrow 00B \Rightarrow 000$
- $A \ast \Rightarrow 000$
- $A \vdash \Rightarrow 000$
A classification of languages based on the form of grammar rules

- Classify not based on how complex the language is
- Classify based on how complex the grammar (the describe the language) is

Four (4) types of grammars:

- Type 0 — recursive grammar
- Type 1 — context sensitive grammar
- Type 2 — context free grammar
- Type 3 — regular grammar
Type 0: Unrestricted/Recursive Grammar

- Type 0 grammar — unrestricted or recursive grammar
  - Form of rules
    \[ \alpha \rightarrow \beta \]
  
  where \( \alpha \in (N \cup T)^+, \beta \in (N \cup T)^* \)
  - No restrictions on form of grammar rules
  - Example:
    - \( aAB \rightarrow aCD \)
    - \( aAB \rightarrow aB \)
    - \( A \rightarrow \varepsilon \) ; \( \varepsilon \)-productions are allowed
  - Exactly set of languages accepted by Turing Machines
  - Derivation strings may contract (LHS may be longer than RHS)
    - Unbounded number of productions before target string
    - Computational complexity: unbounded
Type 1: Context Sensitive Grammar

Type 1 grammar — context sensitive grammar

- Form of rules
  \[ \alpha A \beta \rightarrow \alpha \gamma \beta \]

where \( A \in N \), \( \alpha, \beta \in (N \cup T)^* \), \( \gamma \in (N \cup T)^+ \)

- Replace \( A \) by \( \gamma \) only if found in the context of \( \alpha \) and \( \beta \)
- No \( \epsilon \)-productions allowed
- Example:
  \[ aAB \rightarrow aCB \]
- Derivation strings never contract (LHS always shorter or equal to RHS)
  - Bounded number of productions before target string
  - Computational complexity: NP-Complete
Type 2: Context Free Grammar

- Type 2 grammar — context free grammar
  - Form of rules
    \[ A \rightarrow \gamma \]
  
  where \( A \in N, \gamma \in (N \cup T)^+ \)

- Can replace \( A \) by \( \gamma \) regardless of context
- No \( \varepsilon \)-productions allowed in “proper” CFGs
  - Sometimes relax this restriction to simplify representation
  - Rules always rewritable to exclude \( \varepsilon \)-productions

- Exactly set of languages accepted by Pushdown Automata
- Computational complexity: Polynomial, most real world CFGs are \( O(n) \)
Type 2: Context Free Grammar

- Type 2 grammar — context free grammar
  - Form of rules
    \[ A \rightarrow \gamma \]
    where \( A \in N, \gamma \in (N \cup T)^+ \)
  - Can replace \( A \) by \( \gamma \) regardless of context
  - No \( \varepsilon \)-productions allowed in “proper” CFGs
    - Sometimes relax this restriction to simplify representation
    - Rules always rewritable to exclude \( \varepsilon \)-productions
  - Exactly set of languages accepted by Pushdown Automata
  - Computational complexity: Polynomial, most real world CFGs are \( O(n) \)

- Are programming languages (PLs) context free?
  - Many PL constructs are context free: If-stmt, declaration
  - Some are not: def-before-use, matching formal/actual parameters, etc.
Type 3: Regular Grammar

- Type 3 grammar — regular grammar
  - Form of rules
    \[ A \rightarrow \alpha, \text{ or } A \rightarrow \alpha B \]
    
    where \( A, B \in N, \alpha \in T \)

- Regular grammar defines RE
- Can be used to define tokens for lexical analysis

- Example:
  \[ A \rightarrow 1A \mid 0 \]
Differentiate Type 2 and 3 Grammars

Language $L_1 = \{ [i]^j \mid i, j \geq 1 \}$

- Regular grammar

$$S \rightarrow [S \mid [T$$

$$T \rightarrow ]T \mid ]$$

Language $L_2 = \{ [i]^i \mid i \geq 1 \}$

- Context free grammar

$$S \rightarrow [S] \mid [ ]$$
Differentiate Type 1 and 2 Grammars

- **Type 2 grammar (context free)**

  
  \[
  \begin{align*}
  S & \rightarrow D \ U \\
  D & \rightarrow \text{int } x; \quad \text{int } y; \\
  U & \rightarrow x=1; \quad y=1;
  \end{align*}
  \]

- **Type 1 grammar (context sensitive)**

  
  \[
  \begin{align*}
  S & \rightarrow D \ U \\
  D & \rightarrow \text{int } x; \quad \text{int } y; \\
  \text{int } x; U & \rightarrow \text{int } x; x=1; \\
  \text{int } y; U & \rightarrow \text{int } y; y=1;
  \end{align*}
  \]

What Does a Programming Language Want?

Language from type 2 grammar

- $S \Rightarrow DU \Rightarrow int \ x; \ U \Rightarrow int \ x; \ x=1;$
- $S \Rightarrow DU \Rightarrow int \ x; \ U \Rightarrow int \ x; \ y=1;$
- $S \Rightarrow DU \Rightarrow int \ y; \ U \Rightarrow int \ y; \ x=1;$
- $S \Rightarrow DU \Rightarrow int \ y; \ U \Rightarrow int \ y; \ y=1;$

Language from type 1 grammar

- $S \Rightarrow DU \Rightarrow int \ x; \ U \Rightarrow int \ x; \ x=1;$
- $S \Rightarrow DU \Rightarrow int \ y; \ U \Rightarrow int \ y; \ y=1;$
What Does a Programming Language Want?

Language from type 2 grammar

- $S \Rightarrow DU \Rightarrow \text{int } x; \ U \Rightarrow \text{int } x; \ x=1;$
- $S \Rightarrow DU \Rightarrow \text{int } x; \ U \Rightarrow \text{int } x; \ y=1;$
- $S \Rightarrow DU \Rightarrow \text{int } y; \ U \Rightarrow \text{int } y; \ x=1;$
- $S \Rightarrow DU \Rightarrow \text{int } y; \ U \Rightarrow \text{int } y; \ y=1;$

Language from type 1 grammar

- $S \Rightarrow DU \Rightarrow \text{int } x; \ U \Rightarrow \text{int } x; \ x=1;$
- $S \Rightarrow DU \Rightarrow \text{int } y; \ U \Rightarrow \text{int } y; \ y=1;$

PLs are context sensitive, why use CFG in parsing?
Regular Grammar $\subseteq$ CFG $\subseteq$ CSG $\subseteq$ Recursive Grammar
Regular Grammar $\subseteq$ CFG $\subseteq$ CSG $\subseteq$ Recursive Grammar

However, $L_y \subset L_x$ where $L_x: [i]^k \rightarrow$ RG, $L_y: [i]^i \rightarrow$ CFG

Is it a problem?
Context Free Grammars
Grammar and Syntax Analysis

- Grammar is used to derive string or construct parser

- A derivation is a sequence of applications of rules
  - Starting from the start symbol
  - $S \Rightarrow ... \Rightarrow ... \Rightarrow (sentence)$

- Leftmost and Rightmost derivations
  - At each derivation step, leftmost derivation always replaces the leftmost non-terminal symbol
  - Rightmost derivation always replaces the rightmost one
Examples

E → E * E | E + E | ( E ) | id

- leftmost derivation
  E ⇒ E + E ⇒ E * E + E ⇒ id * E + E ⇒ id * id + E ⇒ ...
  ⇒ id * id + id * id

- rightmost derivation
  E ⇒ E + E ⇒ E + E * E ⇒ E + E * id ⇒ E + id * id ⇒ ...
  ⇒ id * id + id * id
Parse Trees

- Parse tree structure
  - Describes program structure (defined by the rules applied)
  - Internal nodes are non-terminals, leaves are terminals
  - Structure depends on *what* production rules were applied
    - Grammars should specify precisely what rules to apply to avoid ambiguity in program structure
  - Structure does **not** depend on *order* of application
    - Same tree for previous rightmost/leftmost derivations
    - Order of rule application is implementation dependent
Parse Trees

- Parse tree structure
  - Describes program structure (defined by the rules applied)
  - Internal nodes are non-terminals, leaves are terminals
  - Structure depends on **what** production rules were applied
    - Grammars should specify precisely what rules to apply to avoid ambiguity in program structure
  - Structure does **not** depend on **order** of application
    - Same tree for previous rightmost/leftmost derivations
    - Order of rule application is implementation dependent

```
E
  / + \
E   E
  \   \     \   \     \   
E     E   E     E   E
  |     |   |     |   |     |
  id   id   id   id   id
```
Consider the string
\[ \text{id} * \text{id} + \text{id} * \text{id} \]
can result in 3 different trees (and more) for this grammar
A grammar $G$ is **ambiguous** if

- there exist a string $str \in L(G)$ such that
- more than one parse tree derives $str$

In practice, we prefer unambiguous grammars

Fortunately, ambiguity is (often) the property of a grammar and not the language

- It is (often) possible to rewrite grammar to remove ambiguity
- Programming languages are designed not to be ambiguous — it’s the job of compiler writers to express those rules accurately in the grammar
How to Remove Ambiguity?

Method I: to specify **precedence**

- build precedence into grammar, have different non-terminal for each precedence level
  - Lower precedence — relatively higher in tree (close to root)
  - Higher precedence — relatively lower in tree (far from root)
  - Same precedence — depends on associativity

\[
E \rightarrow E + E \mid E - E \mid E \ast E \mid E / E \mid E \land E \mid ( E ) \mid id
\]

rewrite it to

\[
E \rightarrow E + T \mid E - T \mid T
\]
\[
T \rightarrow T \ast F \mid T / F \mid F
\]
\[
F \rightarrow B \land F \mid B
\]
\[
B \rightarrow id \mid ( E )
\]
How to Remove Ambiguity?

- Method II: to specify **associativity**
  - Allow recursion only on either left or right non-terminal
    - Left associative — recursion on left non-terminal
    - Right associative — recursion on right non-terminal

- For the previous example,

  \[ E \rightarrow E + E \ldots \quad ; \text{allows both left/right associativity} \]

  rewrite it to

  \[ E \rightarrow E + T \ldots \quad ; \text{only left associativity} \]
  \[ F \rightarrow B \land F \ldots \quad ; \text{only right associativity} \]
Properties of Context Free Grammars

- **Decidable**: computable using a Turing Machine
  - In other words, guaranteed an answer in finite time

- **If a string is in a context free language**: Decidable
  - Implementing a parser is feasible for every CFL

- **If a CFG is ambiguous**: Undecidable
  - Deciding ambiguity by analyzing grammar is impossible
  - Can only check whether ambiguous for a given string
  - In practice, tools like Yacc check for a more restricted grammar (e.g. LALR(1)) instead
    - LALR(1) is a subset of unambiguous grammars
    - Whether grammar is LALR(1) is easily analyzable

- **If two CFGs generate same language**: Undecidable
  - Impossible to tell if language changed by tweaking grammar
  - Parsers are regression tested against a test set frequently
From Grammar to Parser

- What exactly is parsing, or syntax analysis?
  - To process an input string for a given grammar, and compose the derivation if the string is in the language.

- Two subtasks
  - To determine if string in the language or not
  - To construct the parse tree (a representation of the derivation)
What exactly is parsing, or syntax analysis?

- To process an input string for a given grammar, and compose the derivation if the string is in the language.

- Two subtasks:
  - to determine if string in the language or not
  - to construct the parse tree (a representation of the derivation)

How would you construct a parser from a grammar?
Types of Parsers

- Universal parser
  - Can parse any CFG e.g. Early’s algorithm
  - Powerful but extremely inefficient
    \( O(N^3) \) where \( N \) is length of string

- Top-down parser
  - Tries to expand start symbol to input string
  - Finds leftmost derivation
  - Only works for a certain class of CFGs
  - Starts from root and expands into leaves
  - Structure closely mimics grammar — amenable to implementation by hand
Types of Parsers (cont.)

- **Bottom-up parser**
  - Tries to *reduce* the input string to the start symbol
  - Finds reverse order of the rightmost derivation
  - Works for a wider class of CFGs
  - Starts at leaves and build tree in bottom-up fashion
  - More amenable to generation by an automated tool
What Output do We Want?

- The output of parsing is
  - parse tree, or
  - abstract syntax tree

- An abstract syntax tree is
  - abbreviated representation of a parse tree
  - drops some details without compromising meaning
    - some terminal symbols that no longer contribute to semantics are dropped (e.g. parentheses)
    - internal nodes may contain terminal symbols
Consider the grammar

\[ E \rightarrow \text{int} \mid (E) \mid E + E \]

and an input

\[ 5 + (2 + 3) \]

After lexical analysis, we have a sequence of tokens

\[ \text{INT}_5 \, '+' \, '(' \, \text{INT}_2 \, '+' \, \text{INT}_3 \, ')' \]
A parse tree

- Traces the process of derivation
- Captures all the rules that were applied

but contains redundant information

- parentheses
- single-successor nodes
An Abstract Syntax Tree (AST) for the input

- AST captures all that is semantically relevant
- AST but is more compact in representation
- AST abstracts from parse tree (a.k.a. concrete syntax tree)
- ASTs are used in most compilers rather than parse trees
How are ASTs constructed in a compiler?

Through implementation of **semantic actions**
- Semantic actions: actions triggered on a rule match
- Already used in lexical analysis to return token tuples
- For syntactic analysis, involves computing semantic value of whole construct from values of its parts

To construct AST, attach an **attribute** for each symbol X
- Attributes contain the semantic value for each symbol
  - \( X.\text{ast} \) — the constructed AST for symbol X

Then attach semantic actions to production rules, e.g.

\[
X \rightarrow Y_1 Y_2 \ldots Y_n \quad \{ \text{actions} \}
\]

actions define \( X.\text{ast} \) using \( Y_i.\text{ast} \) (\( 1 \leq i \leq n \))
Example

For the previous example, we have

\[ E \rightarrow \text{int} \quad \{ \text{E.ast} = \text{mkleaf(int.lval)} \} \]
\[ | \quad \text{E1 + E2} \quad \{ \text{E.ast} = \text{mkplus(E1.ast, E2.ast)} \} \]
\[ | \quad (\text{E1}) \quad \{ \text{E.ast} = \text{E1.ast} \} \]

Two functions that need to be defined

- ptr1 = mkleaf(n) — create a leaf node and assign value “n”
- ptr2 = mkplus(t1, t2) — create a tree node and assign value “PLUS”, then attach two child trees t1 and t2
AST Construction Steps

For input $\text{INT}_5 \ ' + ' \ (' \ \text{INT}_2 \ ' + ' \ \text{INT}_3 \ ')$

Construction order given is for a top-down LL(1) parser
(Order can change depending on parser implementation)
For input INT₅ ‘+’ (’ INT₂ ‘+’ INT₃ ‘)’
Construction order given is for a top-down LL(1) parser
(Order can change depending on parser implementation)
AST Construction Steps

For input $\text{INT}_5 \ '+' '(' \text{INT}_2 \ '+' \text{INT}_3 ')'$

Construction order given is for a top-down LL(1) parser
(Order can change depending on parser implementation)

$E1.\text{ast}=\text{mkleaf}(5) \ E2.\text{ast}=\text{mkleaf}(2)$
For input \( \text{INT}_5 \  `+` \ (`\text{INT}_2 \ `+` \ \text{INT}_3 \`) \)
Construction order given is for a top-down LL(1) parser
(Order can change depending on parser implementation)

\[
\text{E1.ast=mkleaf(5)} \ \text{E2.ast=mkleaf(2)} \ \text{E3.ast=mkleaf(3)}
\]
For input \textbf{INT}_5 \, '+\, (\, \textbf{INT}_2 \, '+\, \textbf{INT}_3 \, \,')$

Construction order given is for a top-down LL(1) parser
(Order can change depending on parser implementation)

\begin{itemize}
  \item \textbf{E1}.ast=\textit{mkleaf}(5)
  \item \textbf{E2}.ast=\textit{mkleaf}(2)
  \item \textbf{E3}.ast=\textit{mkleaf}(3)
  \item \textbf{E4}.ast=\textit{mkplus}(\textbf{E2}.ast, \textbf{E3}.ast)
\end{itemize}
AST Construction Steps

For input $\text{INT}_5 \ ' + ' \ ( \ \text{INT}_2 \ ' + ' \ \text{INT}_3 \ )$

Construction order given is for a top-down LL(1) parser
(Order can change depending on parser implementation)

$$E4.\text{ast} = \text{mkplus}(E2.\text{ast}, E3.\text{ast})$$

$$E1.\text{ast} = \text{mkleaf}(5)$$
AST Construction Steps

For input $\text{INT}_5 \, '+\, (' \text{INT}_2 \, '+ \, \text{INT}_3 \, ')'$

Construction order given is for a top-down LL(1) parser
(Order can change depending on parser implementation)

$E_5.\text{ast}=\text{mkplus}(E_1.\text{ast}, \, E_4.\text{ast})$

$E_4.\text{ast}=\text{mkplus}(E_2.\text{ast}, \, E_3.\text{ast})$

$E_1.\text{ast}=\text{mkleaf}(5)$
AST Construction Steps

For input $\text{INT}_5 \ ' + ' ( \ ' + ' \text{INT}_2 \ ' + ' \text{INT}_3 \ ')$

Construction order given is for a top-down LL(1) parser
(Order can change depending on parser implementation)

$E_5.\text{ast}=\text{mkplus}(E_1.\text{ast}, E_4.\text{ast})$
Summary

- Compilers specify program structure using CFG
  - Most programming languages are not context free
  - Context sensitive analysis can easily separate out to semantic analysis phase

- A parser uses CFG to
  - ... answer if an input str $\in L(G)$
  - ... and build a parse tree
  - ... or build an AST instead
  - ... and pass it to the rest of compiler
Parsing
We will study two approaches

- **Top-down**
  - Easier to understand and implement manually

- **Bottom-up**
  - More powerful, can be implemented automatically
Consider a CFG grammar G

\[
\begin{align*}
S & \rightarrow A \ B \\
A & \rightarrow a \ C \\
B & \rightarrow b \ D \\
D & \rightarrow d \\
C & \rightarrow c
\end{align*}
\]

Actually, this language has only one sentence, i.e. \( L(G) = \{ \text{acbd} \} \)

**Leftmost Derivation:**

\[
\begin{align*}
S & \Rightarrow AB \ (1) \\
& \Rightarrow aCB \ (2) \\
& \Rightarrow acB \ (3) \\
& \Rightarrow acbD \ (4) \\
& \Rightarrow acbd \ (5)
\end{align*}
\]

**Rightmost Derivation:**

\[
\begin{align*}
S & \Rightarrow AB \ (5) \\
& \Rightarrow AbD \ (4) \\
& \Rightarrow Abd \ (3) \\
& \Rightarrow aCbd \ (2) \\
& \Rightarrow acbd \ (1)
\end{align*}
\]
Consider a CFG grammar $G$

$S \rightarrow A \ B$
$A \rightarrow a \ C$
$B \rightarrow b \ D$
$D \rightarrow d$
$C \rightarrow c$

Actually, this language has only one sentence, i.e.
$L(G) = \{ \text{acbd} \}$

**Leftmost Derivation:**

$S \Rightarrow AB \ (1)$
$\Rightarrow aCB \ (2)$
$\Rightarrow acB \ (3)$
$\Rightarrow acbD \ (4)$
$\Rightarrow acbd \ (5)$

**Rightmost Derivation:**

$S \Rightarrow AB \ (5)$
$\Rightarrow AbD \ (4)$
$\Rightarrow Abd \ (3)$
$\Rightarrow aCbd \ (2)$
$\Rightarrow acbd \ (1)$
Example

Consider a CFG grammar G

\[ S \rightarrow A \ B \] 
\[ A \rightarrow a \ C \] 
\[ B \rightarrow b \ D \] 
\[ D \rightarrow d \] 
\[ C \rightarrow c \] 

Actually, this language has only one sentence, i.e.

\[ L(G) = \{ \text{acbd} \} \]

**Leftmost Derivation:**

1. \( S \rightarrow AB \)
2. \( aCB \)
3. \( acB \)
4. \( acbD \)
5. \( acbd \)

**Rightmost Derivation:**

5. \( S \rightarrow AB \)
4. \( AbD \)
3. \( Abd \)
2. \( aCbd \)
1. \( acbd \)
Example

Consider a CFG grammar $G$

- $S \rightarrow A\ B$
- $A \rightarrow a\ C$
- $B \rightarrow b\ D$
- $D \rightarrow d$
- $C \rightarrow c$

Actually, this language has only one sentence, i.e.
$L(G) = \{\ acbd \}$

Leftmost Derivation:

1. $S \Rightarrow AB$
2. $aCB$
3. $acB$
4. $acbD$
5. $acbd$

Rightmost Derivation:

1. $S \Rightarrow AB$
2. $aCbd$
3. $Abd$
4. $AbD$
5. $Acbd$
Consider a CFG grammar $G$

$$S \rightarrow A \ B \quad A \rightarrow a \ C \quad B \rightarrow b \ D$$

$$D \rightarrow d \quad C \rightarrow c$$

Actually, this language has only one sentence, i.e.

$L(G) = \{ \text{acbd} \}$

**Leftmost Derivation:**

$S \Rightarrow AB$ (1)

$\Rightarrow aCB$ (2)

$\Rightarrow acB$ (3)

$\Rightarrow acbD$ (4)

$\Rightarrow acbd$ (5)

**Rightmost Derivation:**

$S \Rightarrow AB$ (5)

$\Rightarrow AbD$ (4)

$\Rightarrow Abd$ (3)

$\Rightarrow aCbd$ (2)

$\Rightarrow acbd$ (1)
Example

Consider a CFG grammar $G$

$$S \rightarrow A \ B \quad A \rightarrow a \ C \quad B \rightarrow b \ D$$

$$D \rightarrow d \quad C \rightarrow c$$

Actually, this language has only one sentence, i.e.

$$L(G) = \{ \text{acbd} \}$$

**Leftmost Derivation:**

\[
S \Rightarrow AB \ (1) \\
\Rightarrow aCB \ (2) \\
\Rightarrow acB \ (3) \\
\Rightarrow acbD \ (4) \\
\Rightarrow acbd \ (5)
\]

**Rightmost Derivation:**

\[
S \Rightarrow AB \ (5) \\
\Rightarrow AbD \ (4) \\
\Rightarrow Abd \ (3) \\
\Rightarrow aCbd \ (2) \\
\Rightarrow acbd \ (1)
\]
Consider a CFG grammar $G$

$S \rightarrow A \ B$
$A \rightarrow a \ C$
$B \rightarrow b \ D$
$D \rightarrow d$
$C \rightarrow c$

Actually, this language has only one sentence, i.e. $L(G) = \{ \text{acbd} \}$

Leftmost Derivation:

1. $S \Rightarrow AB$
2. $\Rightarrow aCB$
3. $\Rightarrow acB$
4. $\Rightarrow acbD$
5. $\Rightarrow acbd$

Rightmost Derivation:

5. $S \Rightarrow AB$
4. $\Rightarrow AbD$
3. $\Rightarrow Abd$
2. $\Rightarrow aCbd$
1. $\Rightarrow acbd$
Example

Consider a CFG grammar $G$

$S \rightarrow A \ B$
$A \rightarrow a \ C$
$B \rightarrow b \ D$
$D \rightarrow d$
$C \rightarrow c$

Actually, this language has only one sentence, i.e.
$L(G) = \{ \text{acbd} \}$

**Leftmost Derivation:**

$S \Rightarrow AB$ (1)
$\Rightarrow aCB$ (2)
$\Rightarrow acB$ (3)
$\Rightarrow acbD$ (4)
$\Rightarrow acbd$ (5)

**Rightmost Derivation:**

$S \Rightarrow AB$ (5)
$\Rightarrow AbD$ (4)
$\Rightarrow Abd$ (3)
$\Rightarrow aCbd$ (2)
$\Rightarrow acbd$ (1)
Consider a CFG grammar $G$

$S \rightarrow A\ B$
$A \rightarrow a\ C$
$B \rightarrow b\ D$
$D \rightarrow d$
$C \rightarrow c$

Actually, this language has only one sentence, i.e. $L(G) = \{ \text{acbd} \}$

**Leftmost Derivation:**

$S \Rightarrow AB$ (1)
$\Rightarrow aCB$ (2)
$\Rightarrow acB$ (3)
$\Rightarrow acbD$ (4)
$\Rightarrow acbd$ (5)

**Rightmost Derivation:**

$S \Rightarrow AB$ (5)
$\Rightarrow AbD$ (4)
$\Rightarrow Abd$ (3)
$\Rightarrow aCbd$ (2)
$\Rightarrow acbd$ (1)
Consider a CFG grammar $G$

$S \rightarrow A \ B$ \hspace{1em} $A \rightarrow a \ C$ \hspace{1em} $B \rightarrow b \ D$

$D \rightarrow d$ \hspace{1em} $C \rightarrow c$

Actually, this language has only one sentence, i.e.
$L(G) = \{ \text{acbd} \}$

**Leftmost Derivation:**

1. $S \Rightarrow AB$ (1)
2. $\Rightarrow aCB$ (2)
3. $\Rightarrow acB$ (3)
4. $\Rightarrow acbD$ (4)
5. $\Rightarrow acbd$ (5)

**Rightmost Derivation:**

1. $S \Rightarrow AB$ (5)
2. $\Rightarrow AbD$ (4)
3. $\Rightarrow Abd$ (3)
4. $\Rightarrow aCbd$ (2)
5. $\Rightarrow acbd$ (1)
Example

Consider a CFG grammar G

\[
\begin{align*}
S & \rightarrow A \ B \\
A & \rightarrow a \ C \\
B & \rightarrow b \ D \\
D & \rightarrow d \\
C & \rightarrow c
\end{align*}
\]

Actually, this language has only one sentence, i.e.
\[L(G) = \{ \text{acbd} \} \]

Leftmost Derivation:
\[
\begin{align*}
S & \Rightarrow AB \ (1) \\
& \Rightarrow aCB \ (2) \\
& \Rightarrow acB \ (3) \\
& \Rightarrow acbD \ (4) \\
& \Rightarrow acbd \ (5)
\end{align*}
\]

Rightmost Derivation:
\[
\begin{align*}
S & \Rightarrow AB \ (5) \\
& \Rightarrow AbD \ (4) \\
& \Rightarrow Abd \ (3) \\
& \Rightarrow aCbd \ (2) \\
& \Rightarrow acbd \ (1)
\end{align*}
\]
Example

Consider a CFG grammar $G$

$$
S \rightarrow A\ B \\
A \rightarrow a\ C \\
B \rightarrow b\ D \\
D \rightarrow d \\
C \rightarrow c
$$

Actually, this language has only one sentence, i.e. $L(G) = \{ \text{acbd} \}$

Leftmost Derivation:

$S \Rightarrow AB$ (1)
$\Rightarrow aCB$ (2)
$\Rightarrow acB$ (3)
$\Rightarrow acbD$ (4)
$\Rightarrow acbd$ (5)

Rightmost Derivation:

$S \Rightarrow AB$ (5)
$\Rightarrow AbD$ (4)
$\Rightarrow Abd$ (3)
$\Rightarrow aCbd$ (2)
$\Rightarrow acbd$ (1)
Top Down Parsers

- **Recursive descent parser**
  - Implemented using recursive calls to functions that implement the expansion of each non-terminal
  - Goes through all possible expansions by trial-and-error until match with input; backtracks when mismatch detected
  - Simple to implement, but may take exponential time

- **Predictive parser**
  - Recursive descent parser with prediction (no backtracking)
  - Predict next rule by looking ahead $k$ number of symbols
  - Restrictions on the grammar to avoid backtracking
    - Only works for a class of grammars called $LL(k)$

- **Nonrecursive predictive parser**
  - Predictive parser with no recursive calls
  - Table driven — suitable for automated parser generators
Recursive Descent Example

\[
E \rightarrow T + E \mid T \\
T \rightarrow \text{int} \ast T \mid \text{int} \mid (E)
\]

input string: \( \text{int} \ast \text{int} \)
start symbol: \( E \)

initial parse tree is \( E \)
Recursive Descent Example

\[ E \rightarrow T + E \mid T \]
\[ T \rightarrow \text{int} \ast T \mid \text{int} \mid (E) \]

input string: \text{int} \ast \text{int}

start symbol: \text{E}

initial parse tree is \text{E}

Assume: when there are alternative rules, try right rule first
Parsing Sequence (using Backtracking)

E
Parsing Sequence (using Backtracking)

\[
E \Rightarrow T
\]

– pick right most rule \( E \rightarrow T \)
Parsing Sequence (using Backtracking)

\[ E \Rightarrow T \Rightarrow (E) \]

- pick right most rule \( E \rightarrow T \)
- pick right most rule \( T \rightarrow (E) \)
Parsing Sequence (using Backtracking)

\[ E \Rightarrow T \Rightarrow (E) \]

- pick right most rule \( E \rightarrow T \)
- pick right most rule \( T \rightarrow (E) \)
- "(" does not match "int"
Parsing Sequence (using Backtracking)

\[ E \Rightarrow T \Rightarrow (E) \]

- pick right most rule \( E \rightarrow T \)
- pick right most rule \( T \rightarrow (E) \)
- “(” does not match “int”
- failure, backtrack one level
Parsing Sequence (using Backtracking)

\[ E \Rightarrow T \Rightarrow (E) \]

- pick right most rule \( E \rightarrow T \)
- pick right most rule \( T \rightarrow (E) \)
- “(” does not match “int”
- failure, backtrack one level
Parsing Sequence (using Backtracking)

\[ E \Rightarrow T \Rightarrow (E) \]

- pick right most rule \( E \rightarrow T \)
- pick right most rule \( T \rightarrow (E) \)
- "(" does not match "int"
- failure, backtrack one level
- pick next \( T \rightarrow \text{int} \)
- "int" matches input "int"
Parsing Sequence (using Backtracking)

\[ E \Rightarrow T \Rightarrow (E) \]

- pick right most rule \( E \rightarrow T \)
- pick right most rule \( T \rightarrow (E) \)
- “(” does not match “int”
- failure, backtrack one level
- pick next \( T \rightarrow \text{int} \)
- “int” matches input “int”
- however, more tokens remain
- failure, backtrack one level
Parsing Sequence (using Backtracking)

\[ E \Rightarrow T \Rightarrow (E) \]

- pick right most rule \( E \rightarrow T \)
- pick right most rule \( T \rightarrow (E) \)
- "(" does not match "int"
- failure, backtrack one level
- pick next \( T \rightarrow \text{int} \)
- "int" matches input "int"
- however, more tokens remain
- failure, backtrack one level
Parsing Sequence (using Backtracking)

\[ E \Rightarrow T \Rightarrow (E) \]

- pick right most rule \( E \rightarrow T \)
- pick right most rule \( T \rightarrow (E) \)
- “(” does not match “int”
- failure, backtrack one level

\[ \Rightarrow \text{int} \]

- pick next \( T \rightarrow \text{int} \)
- “int” matches input “int”
- however, more tokens remain
- failure, backtrack one level

\[ \Rightarrow \text{int} \ast \ T \]

- pick next \( T \rightarrow \text{int} \ast \ T \)
- “int *” matches input “int *”
Parsing Sequence (using Backtracking)

\[ E \Rightarrow T \Rightarrow (E) \]

\[ \Rightarrow \text{int} \]

\[ \Rightarrow \text{int} * T \Rightarrow \text{int} * (E) \]

- pick right most rule \( E \rightarrow T \)
- pick right most rule \( T \rightarrow (E) \)
- “(” does not match “int”
- failure, backtrack one level
- pick next \( T \rightarrow \text{int} \)
- “int” matches input “int”
- however, more tokens remain
- failure, backtrack one level
- pick next \( T \rightarrow \text{int} * T \)
- “int *” matches input “int *”
- pick right most rule \( T \rightarrow (E) \)
Parsing Sequence (using Backtracking)

\[ E \Rightarrow T \Rightarrow (E) \]

\[ \Rightarrow \text{int} \]

\[ \Rightarrow \text{int} \times T \Rightarrow \text{int} \times (E) \]

- pick right most rule \( E \rightarrow T \)
- pick right most rule \( T \rightarrow (E) \)
- "(" does not match "int"
- failure, backtrack one level
- pick next \( T \rightarrow \text{int} \)
- "int" matches input "int"
- however, more tokens remain
- failure, backtrack one level
- pick next \( T \rightarrow \text{int} \times T \)
- "int *" matches input "int *"
- pick right most rule \( T \rightarrow (E) \)
- "(" does not match input "int"
- failure, backtrack one level
Parsing Sequence (using Backtracking)

E → T → (E)

⇒ int

⇒ int * T → int * (E)

- pick right most rule E→T
- pick right most rule T→(E)
- “(” does not match “int”
- failure, backtrack one level
- pick next T→int
- “int” matches input “int”
- however, more tokens remain
- failure, backtrack one level
- pick next T→int * T
- “int *” matches input “int *”
- pick right most rule T→(E)
- “(” does not match input “int”
- failure, backtrack one level
Parsing Sequence (using Backtracking)

\[ E \Rightarrow T \Rightarrow (E) \]

- pick right most rule \( E \rightarrow T \)
- pick right most rule \( T \rightarrow (E) \)
- “(” does not match “int”
- failure, backtrack one level

\[ \Rightarrow \text{int} \]

- pick next \( T \rightarrow \text{int} \)
- “int” matches input “int”
- however, more tokens remain
- failure, backtrack one level

\[ \Rightarrow \text{int} \ast T \Rightarrow \text{int} \ast (E) \]

- pick next \( T \rightarrow \text{int} \ast T \)
- “int *” matches input “int *”
- pick right most rule \( T \rightarrow (E) \)
- “(” does not match input “int”
- failure, backtrack one level

\[ \Rightarrow \text{int} \ast \text{int} \]

- pick next \( T \rightarrow \text{int} \)
Parsing Sequence (using Backtracking)

\[ E \rightarrow T \rightarrow (E) \]

- pick right most rule \( E \rightarrow T \)
- pick right most rule \( T \rightarrow (E) \)
- “(” does not match “int”
- failure, backtrack one level

\[ \rightarrow \text{int} \]

- pick next \( T \rightarrow \text{int} \)
- “int” matches input “int”
- however, more tokens remain
- failure, backtrack one level

\[ \rightarrow \text{int * T} \rightarrow \text{int * (E)} \]

- pick next \( T \rightarrow \text{int * T} \)
- “int *” matches input “int *”
- pick right most rule \( T \rightarrow (E) \)
- “(” does not match input “int”
- failure, backtrack one level

\[ \rightarrow \text{int * int} \]

- pick next \( T \rightarrow \text{int} \)
- match, accept
Recursive Descent Parsing uses Backtracking

- **Approach**: for a non-terminal in the derivation, productions are tried in some order until
  - A production is found that generates a portion of the input, or
  - No production is found that generates a portion of the input, in which case backtrack to previous non-terminal

- Terminals of the derivation are compared against input
  - Match — advance input, continue parsing
  - Mismatch — backtrack, or fail

- Parsing fails if no production for the start symbol generates the entire input
Create a procedure for each non-terminal

1. For RHS of each production rule,
   a. For a terminal, match with input symbol and consume
   b. For a non-terminal, call procedure for that non-terminal
   c. If match succeeds for entire RHS, return success
   d. If match fails, regurgitate input and try next production rule

2. If match succeeds for any rule, apply that rule to LHS
Sample Code

Sample implementation of parser for previous grammar:

\[E \rightarrow T + E \mid T\]
\[T \rightarrow \text{int} \ast T \mid \text{int} \mid (E)\]

```c
void term()
{
    if (sym==IntNum) {
        fetchNext();
        if (sym==StarNum) {
            fetchNext();
            term();
        }
    } else if (sym==LeftParenNum) {
        fetchNext();
        expr();
        if (sym==RightParenNum)
            fetchNext();
    }
}
```

```c
void expr()
{
    term();
    if (sym==AddNum) {
        fetchNext();
    }
}
```

```c
fetchNext()
{
    ...
}
```
Left Recursion Problem

- The previous scheme does not work if grammar is left recursive
  - Right recursion is okay

- Why is left recursion a problem?
  - For left recursive grammar
    \[
    A \rightarrow A \ b \ | \ c
    \]
  - We may repeatedly choose to apply \( A \ b \)
    \[
    A \Rightarrow A \ b \Rightarrow A \ b \ b \ ...
    \]
  - Sentence can grow indefinitely w/o consuming input
  - How do you know when to stop recursion and choose \( c \)?
The previous scheme does not work if grammar is left recursive

- Right recursion is okay

Why is left recursion a problem?

- For left recursive grammar
  \[ A \rightarrow A \ b \mid c \]
  
  - We may repeatedly choose to apply \( A \ b \)
    \[ A \Rightarrow A \ b \Rightarrow A \ b \ b \ldots \]
  
  - Sentence can grow indefinitely w/o consuming input
  
  - How do you know when to stop recursion and choose \( c \)?

Rewrite the grammar so that it is right recursive

- Which expresses the same language
Remove Left Recursion

In general, we can eliminate all immediate left recursion

\[ A \rightarrow A \cdot x \mid y \]

change to

\[ A \rightarrow y \cdot A' \]
\[ A' \rightarrow x \cdot A' \mid \varepsilon \]

Not all left recursion is immediate

may be hidden in multiple production rules

\[ A \rightarrow BC \mid D \]
\[ B \rightarrow AE \mid F \]

... see Section 4.3 for elimination of general left recursion

... (not required for this course)
Recursive descent is a simple and general parsing strategy.

- Left-recursion must be eliminated first
  - Can be eliminated automatically as in previous slide

However, it is not popular because of backtracking.

- Backtracking requires re-parsing the same string
- Which is inefficient (can take exponential time)
- Also undoing semantic actions may be difficult
  - E.g. removing already added nodes in parse tree

Techniques used in practice do no backtracking...

... at the cost of restricting the class of grammar.
To avoid backtracking: for each non-terminal, choose the appropriate production given next input terminal

- If first terminal of every alternative production is unique
  
  \[
  A \rightarrow a \ B \ D \mid b \ B \ B \\
  B \rightarrow c \mid b \ c \ e \\
  D \rightarrow d
  \]

  parsing input “abcd” requires no backtracking
Predictive Parsers

- To avoid backtracking: for each non-terminal, choose the appropriate production given next input terminal
  - If first terminal of every alternative production is unique
    ```
    A → a B D | b B B
    B → c | b c e
    D → d
    ```
    Parsing input “abced” requires no backtracking

- Rewriting grammars for predictive parsing
  - Left factor to enable prediction
    ```
    A → αβ | αγ
    ```
    Can be changed to
    ```
    A → α A’
    A’ → β | γ
    ```
  - Eliminate left recursion, just like for recursive descent
LL(k) Parsers

- **LL(k) Parser**
  - L — left to right scan
  - L — leftmost derivation
  - k — k symbols of lookahead
  - A predictive parser that uses k lookahead tokens

- **LL(k) Grammar**
  - A grammar that can parsed using a LL(k) parser with no backtracking

- **LL(k) Language**
  - A language that can be expressed as a LL(k) grammar
  - LL(k) languages are a restricted subset of CFLs
  - But many languages are LL(k).. in fact many are LL(1)!

- Can be implemented in a recursive or nonrecursive fashion
Nonrecursive Predictive Parser

Syntax stack — holds unconsumed portion of derivation string
Parse table $M[A,b]$ — an entry containing rule “$A \rightarrow ...$” or error
Parser driver — next action based on (current token, stack top)
Table-driven: amenable to automatic code generation (just like lexers)
## A Sample Parse Table

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E → TX</td>
<td>E → TX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X → +E</td>
<td>X → ε</td>
<td>X → ε</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T → int Y</td>
<td>T → (E)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y → *T</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td></td>
</tr>
</tbody>
</table>
Algorithm for Parsing

X — symbol at the top of the syntax stack
a — current input symbol

 Parsing based on \((X,a)\)

- If \(X==a==\$\), then
  - parser halts with “success”
- If \(X==a!=\$\), then
  - pop \(X\) from stack \textbf{and} advance input head
- If \(X!=a\), then
  Case (a): if \(X \in T\), then
    - parser halts with “failed”, input rejected
  Case (b): if \(X \in N\), \(M[X,a] = “X \rightarrow \text{RHS}”\)
    - pop \(X\) \textbf{and} push \(\text{RHS}\) to stack in reverse order
Push RHS in Reverse Order

X — symbol at the top of the syntax stack
a — current input symbol

if $M[X,a] = "X \rightarrow B \ c \ D"$

\[
\begin{array}{c}
X \\
\ldots \\
$
\end{array} \\
\Rightarrow \\
\begin{array}{c}
\ \ \ \ \ \ \ \ \ \ \ B \\
C \\
D \\
\ldots \\
$
\end{array}
\]
Applying LL(1) Parsing to a Grammar

Given our old grammar

\[
\begin{align*}
E & \rightarrow T + E \mid T \\
T & \rightarrow \text{int} \ast T \mid \text{int} \mid (E)
\end{align*}
\]

- No left recursion
- But require left factoring

After rewriting grammar, we have

\[
\begin{align*}
E & \rightarrow T X \\
X & \rightarrow + E \mid \epsilon \\
T & \rightarrow \text{int} Y \mid (E) \\
Y & \rightarrow \ast T \mid \epsilon
\end{align*}
\]
Using the Parse Table

To recognize “int * int”

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
</tr>
<tr>
<td>$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table</th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E → TX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>X → +E</td>
<td></td>
<td>X → ε</td>
<td>X → ε</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T → int Y</td>
<td></td>
<td></td>
<td>T → (E)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>Y → *T</td>
<td>Y → ε</td>
<td></td>
<td>Y → ε</td>
<td>Y → ε</td>
</tr>
</tbody>
</table>
To recognize “int * int”

Using the Parse Table

Stack

<table>
<thead>
<tr>
<th>input</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int $</td>
</tr>
</tbody>
</table>

Table

<table>
<thead>
<tr>
<th>Stack</th>
<th>Table</th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E → TX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>X</td>
<td>X → +E</td>
<td></td>
<td></td>
<td></td>
<td>X → ε</td>
<td>X → ε</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T → int Y</td>
<td></td>
<td></td>
<td></td>
<td>T → (E)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y → *T</td>
<td>Y → ε</td>
<td></td>
<td></td>
<td>Y → ε</td>
<td>Y → ε</td>
<td></td>
</tr>
</tbody>
</table>
To recognize “int * int”

Using the Parse Table

<table>
<thead>
<tr>
<th>Stack</th>
<th>T</th>
<th>X</th>
<th>$</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>parser driver</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Table</th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E → TX</td>
<td></td>
<td></td>
<td>E → TX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X → +E</td>
<td>X → ε</td>
<td>X → ε</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T → int Y</td>
<td>T → (E)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y → *T</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td></td>
</tr>
</tbody>
</table>
Using the Parse Table

To recognize “int * int”

input

<table>
<thead>
<tr>
<th>input</th>
<th>int</th>
<th>*</th>
<th>int</th>
<th>$</th>
</tr>
</thead>
</table>

Stack

<table>
<thead>
<tr>
<th>T</th>
<th>X</th>
<th>$</th>
</tr>
</thead>
</table>

Table

<table>
<thead>
<tr>
<th>Table</th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E → TX</td>
<td></td>
<td>E → TX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X → +E</td>
<td>X → ε</td>
<td>X → ε</td>
<td>X → ε</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T → int Y</td>
<td></td>
<td>T → (E)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y → *T</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td></td>
</tr>
</tbody>
</table>
Using the Parse Table

To recognize “int * int”

```
<table>
<thead>
<tr>
<th>Stack</th>
<th>Table</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td>*</td>
<td>+</td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>Y</td>
<td>E</td>
<td>E \rightarrow TX</td>
<td>E \rightarrow TX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>\rightarrow +E</td>
<td>X \rightarrow \varepsilon</td>
<td>X \rightarrow \varepsilon</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T \rightarrow int Y</td>
<td>T \rightarrow (E)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y \rightarrow *T</td>
<td>Y \rightarrow \varepsilon</td>
<td>Y \rightarrow \varepsilon</td>
<td>Y \rightarrow \varepsilon</td>
<td></td>
</tr>
</tbody>
</table>
```
Using the Parse Table

To recognize “int * int”

<table>
<thead>
<tr>
<th>Stack</th>
<th>input</th>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>table</th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E → TX</td>
<td></td>
<td>E → TX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X → +E</td>
<td></td>
<td>X → ε</td>
<td>X → ε</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T → int Y</td>
<td></td>
<td>T → (E)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y → *T</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Using the Parse Table

To recognize “int * int”

```
input

Table

<table>
<thead>
<tr>
<th>Stack</th>
<th>Table</th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E</td>
<td>int</td>
<td>*</td>
<td>+</td>
<td>(</td>
<td>)</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>+E</td>
<td></td>
<td>X</td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>T</td>
<td>int</td>
<td>Y</td>
<td>T</td>
<td>(E)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y</td>
<td>*T</td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
</tr>
</tbody>
</table>

parser driver

Stack

Y
X
$
Using the Parse Table

To recognize “int * int”

Table:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Table</th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E</td>
<td>E → TX</td>
<td></td>
<td></td>
<td>E → TX</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X → +E</td>
<td></td>
<td>X → ε</td>
<td>X → ε</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>T → int Y</td>
<td></td>
<td>T → (E)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y → *T</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Parser driver:

Stack:
- *
- T
- X
- $

Input:
- int
- *
- int
- $

Parser driver:

Stack:
- *
- T
- X
- $

Table:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Table</th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E</td>
<td>E → TX</td>
<td></td>
<td></td>
<td>E → TX</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X → +E</td>
<td></td>
<td>X → ε</td>
<td>X → ε</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>T → int Y</td>
<td></td>
<td>T → (E)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y → *T</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
To recognize “int * int”

<table>
<thead>
<tr>
<th>Table</th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E → TX</td>
<td></td>
<td></td>
<td>E → TX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X → +E</td>
<td>X → ε</td>
<td>X → ε</td>
<td>X → ε</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T → int Y</td>
<td>T → (E)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y → *T</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td></td>
</tr>
</tbody>
</table>
To recognize “int * int”

Using the Parse Table

**input**

```
int  *  int  $
```

**Stack**

```
T
X
$
```

**Table**

```
<table>
<thead>
<tr>
<th>Table</th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E → TX</td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>ε</td>
<td>X</td>
</tr>
<tr>
<td>T</td>
<td>T → int Y</td>
<td></td>
<td></td>
<td>T</td>
<td>(E)</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y → *T</td>
<td></td>
<td>Y</td>
<td>ε</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Using the Parse Table

To recognize “int * int”

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E → TX</td>
<td>+</td>
<td>E → TX</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>X → +E</td>
<td>X → ε</td>
<td>X → ε</td>
</tr>
<tr>
<td>T</td>
<td>T → int Y</td>
<td></td>
<td>T → (E)</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y → *T</td>
<td>Y → ε</td>
<td></td>
<td>Y → ε</td>
</tr>
</tbody>
</table>
```

Stack:

- int
- Y
- X
- $

Parser driver:

- input
  - int
  - *
  - int
  - $

Table:

- int
- *
- +
- (  
- )
- $

Parser driver:

- int
- *
- int
- $

Stack:

- int
- Y
- X
- $
To recognize “int * int”

```
Table    | int | *   | +   | (   | )   | $    
---------|-----|-----|-----|-----|-----|------
E        | E → TX | E → TX |     |     |     |      
X        | X → +E | X → ε | X → ε |     |     |      
T        | T → int Y | T → (E) |     |     |     |      
Y        | Y → *T | Y → ε | Y → ε | Y → ε | Y → ε |      
```

Stack:

- X
- Y
- $
To recognize “int * int”

Parser driver

Stack

<table>
<thead>
<tr>
<th>Table</th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E → TX</td>
<td></td>
<td></td>
<td>E → TX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X → +E</td>
<td>X → ε</td>
<td>X → ε</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T → int Y</td>
<td>T → (E)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y → *T</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input

Table: To recognize “int * int”
To recognize “int * int”

Using the Parse Table

Table

<table>
<thead>
<tr>
<th>Table</th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E → TX</td>
<td></td>
<td>E → TX</td>
<td></td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>X → +E</td>
<td></td>
<td>X → ε</td>
<td>X → ε</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T → int Y</td>
<td></td>
<td>T → (E)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y → *T</td>
<td>Y → ε</td>
<td></td>
<td>Y → ε</td>
<td>Y → ε</td>
<td></td>
</tr>
</tbody>
</table>

Stack

X
$

input

int * int $
To recognize “int * int”

Using the Parse Table

input

<table>
<thead>
<tr>
<th>Stack</th>
<th>Table</th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>E</td>
<td>E → TX</td>
<td>E → TX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X → +E</td>
<td>X → ε</td>
<td>X → ε</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T → int Y</td>
<td>T → (E)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y → *T</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td>Y → ε</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Using the Parse Table

To recognize “int * int”

input

Stack

$  

Table

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E → TX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>X</td>
<td>X → +E</td>
<td>X → ε</td>
<td>X → ε</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T → int Y</td>
<td></td>
<td>T → (E)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y → *T</td>
<td>Y → ε</td>
<td></td>
<td>Y → ε</td>
<td>Y → ε</td>
<td></td>
</tr>
</tbody>
</table>

parser driver
Using the Parse Table

To recognize “int * int”

```plaintext
input

```
```

Accept!!!
### Recognition Sequence

It is possible to write in a action list

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>E $</td>
<td>int * int $</td>
<td>E → TX</td>
</tr>
<tr>
<td>T X $</td>
<td>int * int $</td>
<td>T → int Y</td>
</tr>
<tr>
<td>int Y X $</td>
<td>int * int $</td>
<td>terminal</td>
</tr>
<tr>
<td>Y X $</td>
<td>* int $</td>
<td>Y → * T</td>
</tr>
<tr>
<td>* T X $</td>
<td>* int $</td>
<td>terminal</td>
</tr>
<tr>
<td>T X $</td>
<td>int $</td>
<td>T → int Y</td>
</tr>
<tr>
<td>int Y X $</td>
<td>int $</td>
<td>terminal</td>
</tr>
<tr>
<td>Y X $</td>
<td>$</td>
<td>Y → ε</td>
</tr>
<tr>
<td>X $</td>
<td>$</td>
<td>X → ε</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>halt and accept</td>
</tr>
</tbody>
</table>
How to Construct the Parse Table?

- Need to know 2 sets
  - For each symbol A, the set of terminals that can begin a string derived from A. This set is called the **FIRST** set of A.
  - For each non-terminal A, the set of terminals that can appear after a string derived from A is called the **FOLLOW** set of A.
Intuitive Meaning of First and Follow

Why is the Follow Set important?
First($\alpha$)

First($\alpha$) = set of terminals that start string of terminals derived from $\alpha$.

Apply following rules until no terminal or $\varepsilon$ can be added

1). If $t \in T$, then First($t$)={$t$}.
   For example First($+$)={$+\}$.

2). If $X \in N$ and $X \rightarrow \varepsilon$ exists, then add $\varepsilon$ to First($X$).
   For example, First($Y$) =$\{*,\varepsilon\}$.

3). If $X \in N$ and $X \rightarrow Y_1 Y_2 Y_3... Y_m$, then
   - Add (First($Y_1$) - $\varepsilon$) to First($X$).
   - If First($Y_1$), ..., First($Y_{k-1}$) all contain $\varepsilon$, then add $(\sum_{1 \leq i \leq k} First(Y_i) - \varepsilon)$ to First($X$).
   - If First($Y_1$), ..., First($Y_m$) all contain $\varepsilon$, then add $\varepsilon$ to First($X$).
Follow($\alpha$)

Follow($\alpha$) = \{ $t | S \Rightarrow ^* \alpha t \beta $ \}

Intuition: if $X \rightarrow A \ B$, then $\text{First}(B) \subseteq \text{Follow}(A)$

little trickier because $B$ may be $\varepsilon$ i.e. $B \Rightarrow ^* \varepsilon$

Apply following rules until no terminal or $\varepsilon$ can be added

1). $\$ \in \text{Follow}(S)$, where $S$ is the start symbol.
   e.g. $\text{Follow}(E) = \{ \$ \ldots \}$.

2). Look at the occurrence of a non-terminal on the right hand side of a production which is followed by something
   If $A \rightarrow \alpha B \beta$, then $\text{First}(\beta) - \{ \varepsilon \} \subseteq \text{Follow}(B)$

3). Look at $N$ on the RHS that is not followed by anything,
   if $(A \rightarrow \alpha B)$ or $(A \rightarrow \alpha B \beta \text{ and } \varepsilon \in \text{First}(\beta))$
   then $\text{Follow}(A) \subseteq \text{Follow}(B)$
Informal Interpretation of First and Follow Sets

- **First** set of $X$ (focus on LHS)
  - If $X \in T$, then terminal symbol itself
  - If $X \rightarrow Y Z$, then add $\text{First}(Y)$
  - If $X \rightarrow \varepsilon$, then add $\varepsilon$

- **Follow** set of $X$ (focus on RHS)
  - If $X$ is start symbol, add $\$$
  - If $... \rightarrow X Y$, then add $\text{First}(Y)$
  - If $Y \rightarrow X$, then add $\text{Follow}(Y)$
For the example

\[
E \rightarrow T \ X \\
X \rightarrow + \ E \mid \varepsilon \\
T \rightarrow \text{int} \ Y \mid ( E ) \\
Y \rightarrow * \ T \mid \varepsilon
\]

First set (focus on LHS)

E \rightarrow T \ X \\
X \rightarrow + \ E \\
X \rightarrow \varepsilon \\
T \rightarrow \text{int} \ Y \\
T \rightarrow ( E ) \\
Y \rightarrow * \ T \\
Y \rightarrow \varepsilon

Follow set (focus on RHS)

$ \rightarrow \\
T \rightarrow ( E ) \\
X \rightarrow + \ E \\
E \rightarrow T \ X \\
Y \rightarrow * \ T \\
T \rightarrow \text{int} \ Y
Example

\[
E \rightarrow T X \\
X \rightarrow + E \mid \varepsilon \\
T \rightarrow \text{int} Y \mid ( E ) \\
Y \rightarrow * T \mid \varepsilon
\]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>First</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
</tr>
<tr>
<td>)</td>
<td>)</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>Y</td>
<td>*, \varepsilon</td>
</tr>
<tr>
<td>X</td>
<td>+, \varepsilon</td>
</tr>
<tr>
<td>T</td>
<td>(, int</td>
</tr>
<tr>
<td>E</td>
<td>(, int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>$,)</td>
</tr>
<tr>
<td>X</td>
<td>$,)</td>
</tr>
<tr>
<td>T</td>
<td>$,),+</td>
</tr>
<tr>
<td>Y</td>
<td>$,),+</td>
</tr>
</tbody>
</table>
To construct the parse table, we check each $A \rightarrow \alpha$

- For each terminal $a \in \text{First}(\alpha)$, then add $A \rightarrow \alpha$ to $M[A, a]$.
- If $\varepsilon \in \text{First}(\alpha)$, then for each terminal $b \in \text{Follow}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$.
- If $\varepsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$, then add $A \rightarrow \alpha$ to $M[A, \$]$. 
Example

\[ E \rightarrow T \ X \]
\[ X \rightarrow + \ E \mid \varepsilon \]
\[ T \rightarrow \text{int} \ Y \mid ( \ E ) \]
\[ Y \rightarrow * \ T \mid \varepsilon \]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>First</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
</tr>
<tr>
<td>)</td>
<td>)</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>Y</td>
<td>*, \varepsilon</td>
</tr>
<tr>
<td>X</td>
<td>+, \varepsilon</td>
</tr>
<tr>
<td>T</td>
<td>(, \text{int}</td>
</tr>
<tr>
<td>E</td>
<td>(, \text{int}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>$, )</td>
</tr>
<tr>
<td>X</td>
<td>$, )</td>
</tr>
<tr>
<td>T</td>
<td>$, ), +</td>
</tr>
<tr>
<td>Y</td>
<td>$, ), +</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table</th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E \rightarrow TX</td>
<td>E \rightarrow TX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X \rightarrow +E</td>
<td>X \rightarrow \varepsilon</td>
<td>X \rightarrow \varepsilon</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T \rightarrow \text{int} \ Y</td>
<td>T \rightarrow (E)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y \rightarrow *T</td>
<td>Y \rightarrow \varepsilon</td>
<td>Y \rightarrow \varepsilon</td>
<td>Y \rightarrow \varepsilon</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Determine if Grammar G is LL(1)

- **Observation**
  If a grammar is LL(1), then each of its LL(1) table entry contains at most one rule. Otherwise, it is not LL(1).

- **Two methods** to determine if a grammar is LL(1) or not
  1. Construct LL(1) table, and check if there is a multi-rule entry or
  2. Checking each rule as if the table is getting constructed.

  G is LL1(1) iff for a rule $A \rightarrow \alpha \mid \beta$
  - $\text{First}(\alpha) \cap \text{First}(\beta) = \phi$
  - at most one of $\alpha$ and $\beta$ can derive $\varepsilon$
  - If $\beta$ derives $\varepsilon$, then $\text{First}(\alpha) \cap \text{Follow}(A) = \phi$
Non-LL(1) Grammars

If an LL(1) table entry contains more than one rule, then the grammar is not LL(1). What to do then?

(1) The *language* may still be LL(1). Try to Massage grammar to LL(1) grammar:
- Apply left-factoring
- Apply left-recursion removal

(2) If (1) fails, the possibilities are...
- Grammar is ambiguous (multiple parse trees)
  - Remove ambiguity by applying precedence and associativity rules, then try to make grammar LL(1)
- Language is not LL(1)
  - Impossible to resolve conflict at the grammar level
  - Allow programmer to choose one rule when generating table (if choosing that rule is always semantically correct)
  - Otherwise, use a more powerful parser (e.g. LL(k), LR(1))
Some grammars are not LL(1) even after left-factoring and left-recursion removal

\[ S \rightarrow \text{if } C \text{ then } S \mid \text{if } C \text{ then } S \text{ else } S \mid \text{a (other statements)} \]
\[ C \rightarrow b \]

change to

\[ S \rightarrow \text{if } C \text{ then } S \ X \mid a \]
\[ X \rightarrow \text{else } S \mid \epsilon \]
\[ C \rightarrow b \]

**problem sentence: “if b then if b then a else a”**

“else” \(\in\) First(X)
\[ \text{First}(X) - \epsilon \subseteq \text{Follow}(S) \]
\[ X \rightarrow \text{else } \cdots \mid \epsilon \]

“else” \(\in\) Follow(X)

Such grammars are potentially ambiguous
Non-LL(1) Even After Removing Ambiguity

For the “if-then-else” example, how would you rewrite it?

\[
S \rightarrow \text{if } C \text{ then } S \mid \text{if } C \text{ then } S2 \text{ else } S \\
S2 \rightarrow \text{if } C \text{ then } S2 \text{ else } S2 \mid a \\
C \rightarrow b
\]

Now grammar is unambiguous but it is not LL(k) for any k

- Must lookahead until matching ‘else’ to choose rule for ‘S’
- That lookahead may be an arbitrary number of tokens with arbitrary nesting of if-then-else statements

Some languages are not LL(1) or even LL(k)

- No amount of lookahead can tell parser whether this statement is an if-then or if-then-else

Fortunately in this case, can resolve conflicts in table

- Always choose \( X \rightarrow \text{else } S \) over \( X \rightarrow \varepsilon \) on ‘else’ token
- Semantically correct, since else should match closest if
LL(1) Time and Space Complexity

- LL(1) parsers operate in linear time and at most linear space relative to the length of input because
  - Time — each input symbol is processed constant number of times
    - Why?
  - Stack space is smaller than the input (if we remove $X \rightarrow \varepsilon$)
    - Why?
Nonrecursive Parser Summary

- **First** and **Follow** sets are used to construct predictive parsing tables

- Intuitively, **First** and **Follow** sets guide the choice of rules
  
  - For non-terminal **A** and lookahead **t**, use the production rule \( A \rightarrow \alpha \) where \( t \in \text{First}(\alpha) \)

  OR

  - For non-terminal **A** and lookahead **t**, use the production rule \( A \rightarrow \alpha \) where \( \varepsilon \in \text{First}(\alpha) \) and \( t \in \text{Follow}(A) \)

  - There can only be ONE such rule or grammar is not LL(1)
Questions

- What is LL(0)?

- Why LL(2) ... LL(k) are not widely used?