Syntax Analysis

Syntax Analysis is the second phase of compilation

- Comparison with lexical analysis:

Phase	Input	Output
Lexer	string of characters	string of tokens
Parser	string of tokens	Parse tree/AST

- Syntax analysis is also called **parsing**
  - Because it produces a parse tree.
  - AST (Abstract Syntax Tree) is a simplified parse tree.

What is a Parse Tree?

- **Parse tree**: a tree that represents grammatical structure
- Language constructs often have recursive structures

  If-stmt $\equiv$ **if** (EXPR) **then Stmt else Stmt fi**

  Stmt $\equiv$ **If-stmt** | **While-stmt** | ...

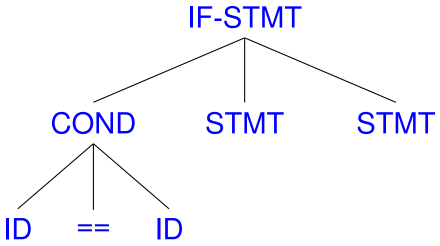
A Parse Tree Example

- Code to be compiled:

  ... if x==y then .... else ... fi

- Lexer: ... ...
- Parser:
  - Input: sequence of tokens

    ... IF ID==ID THEN ... ELSE ... FI
  - Desired output:



REs cannot express recursive program constructs

- Example of a recursive construct is matching parenthesis: # of “(” must equal # of “)”
  - ✔ (x+y)*z
  - ✔ ((x+y)+y)*z
  - ...
  - ✔ (...(((x+y)+y)+y)...)
  - ✘ ((x+y)+y)+y)*z

REs cannot express recursive program constructs

- Example of a recursive construct is matching parenthesis: # of "(" must equal # of ")"
  - ✔ (x+y)*z
  - ✔ ((x+y)+y)*z

    ...
  - ✔ (...(((x+y)+y)+y)...)

  - ✘ ((x+y)+y)+y)*z

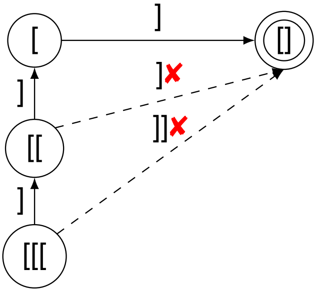
- Can regular expressions express this construct?
  - ➢ Recall RL $\equiv$ L(Regular Expression) $\equiv$ L(Finite Automata)
  - ➢ Boils down to whether an FA can accept this construct

RE/FA is Not Powerful Enough

- Describe strings with pattern $[^i]^i$ **(i≥1)**

RE/FA is Not Powerful Enough

- Describe strings with pattern $[^i]^i$ **(i≥1)**



- ➢ "[", "[[" are different states as only former accepts on "]"
- ➢ "[[", "[[[" are different states as only former accepts on "]]"
- ➢ Infinite as for any $[^i$, there exists a $[^{i+1}$ that is a new state
- ➢ Contradiction: no finite automaton accepts arbitrary nesting

REs are not suitable for Syntax Analysis

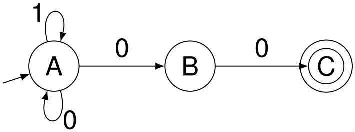
- REs cannot express recursive language constructs
- Programming languages belong to a category called CFLs
  - CFL is short for Context Free Language
  - CFLs are a strictly larger set than RLs
- To express CFLs, we need a new formalism: Grammars
- Grammars are general enough to express most languages
  - Regular Languages
  - Context Free Languages
  - Context Sensitive Languages
  - Recursively Enumerable Languages

A Grammar defines a Language

- A grammar, along with tokens, defines a language
  - Like how English grammar defines the English language
- Grammars are defined using rigorous math just like for REs
- Recall the following definitions
  - Language: A set of strings over alphabet
    Alphabet: A finite set of symbols
    Empty string: ε
- We will also start calling strings in the language **sentences**

An Example Grammar

- Language L = { any string with "00" at the end }



- Grammar G = { T, N, s, δ }
  where T = { 0, 1 }, N = { A, B }, s = A, and
  production rules δ = { A$\rightarrow$ 0A | 1A | 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 $\Rightarrow$ ...

Grammar, formally defined

- A **grammar** consists of 4 components **(T, N, S, δ)**
  - T — set of **terminal** symbols
    - Leaves in the parse tree — essentially tokens
  - N — set of **non-terminal** symbols
    - Internal nodes in the parse tree that expands into tokens
    - Language construct composed of one or more tokens like: statements, loops, functions, classes, ...
  - S — A special non-terminal **start symbol**
    - Every string in language is derived from it
  - δ — a set of **production** rules
    - "LHS $\rightarrow$ RHS": left-hand-side *produces* right-hand-side

Production Rule and Derivation

- "LHS $\rightarrow$ RHS"
  - Production rule to replace LHS with RHS
  - Applied repeatedly to derive target sentence from **S**

- $\beta \Rightarrow \alpha$: string β derives α
  - $\beta \Rightarrow \alpha$ — 1 step
  - $\beta \Rightarrow *\alpha$ — 0 or more steps
  - $\beta \stackrel{*}{\Longrightarrow} \alpha$ — 0 or more steps
  - example:

    A $\Rightarrow$ 0A $\Rightarrow$ 00B $\Rightarrow$ 000

    A $\stackrel{*}{\Longrightarrow}$ 000

    A $\stackrel{+}{\Longrightarrow}$ 000

Noam Chomsky Grammars

- Chomsky classified grammars into 4 types:
  Type 0: recursive grammar
  Type 1: context sensitive grammar
  Type 2: context free grammar
  Type 3: regular grammar
  (Classification done based on form of production rules)

- The grammars produce the corresponding languages:
  L(recursive grammar) $\equiv$ recursively enumerable language
  L(context sensitive grammar) $\equiv$ context sensitive language
  L(context free grammar) $\equiv$ context free language
  L(regular grammar) $\equiv$ regular language

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 $\rightarrow$ ε ; erase rule is allowed

Type 1: Context Sensitive Grammar

- Type 1 grammar — context sensitive grammar
  - Form of rules

    $\alpha \boldsymbol{A} \beta \rightarrow \alpha \gamma \beta$

    where $A \in N$, $\alpha, \beta \in (N \cup T)^*$, $\gamma \in (N \cup T)^+$
  - Replace A by γ only if found in the context of α and β
  - No erase rule
  - Example:

    aAB $\rightarrow$ aCB

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 γ at any time — cannot specify context

Type 2: Context Free Grammar

- Type 2 grammar — context free grammar
  - Form of rules

    $$A \to \gamma$$

    where $A \in N$, $\gamma \in (N \cup T)^+$
  - Can replace A by γ at any time — cannot specify context
- Are programming languages (PLs) context free ?
  - Some PL constructs are context free: If-stmt, declaration
  - Many are not: **def-before-use**, **matching formal/actual parameters**, etc.

Type 3: Regular Grammar

- Type 3 grammar — regular grammar
  - Form of rules

    $A \rightarrow \alpha$, 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 L1 = $\{ [^i]^j \mid i, j >= 1\}$
  - Regular grammar

    S → [S | [T
    T →] T |]

- Language L2 = $\{ [^i]^i \mid i >= 1\}$
  - Context free grammar

    S → [S] | [ ]

Differentiate Type 1 and 2 Grammars

- Type 2 grammar (context free)

$$
\begin{aligned}
&S \rightarrow D\ U \\
&D \rightarrow \text{int x;} \quad | \quad \text{int y;} \\
&U \rightarrow \text{x=1;} \quad | \quad \text{y=1;}
\end{aligned}
$$

- Type 1 grammar (context sensitive)

$$
\begin{aligned}
&S \rightarrow D\ U \\
&D \rightarrow \text{int x;} \quad | \quad \text{int y;} \\
&\text{int x; } U \rightarrow \text{int x; x=1;} \\
&\text{int y; } U \rightarrow \text{int y; y=1;}
\end{aligned}
$$

Are Programming Languages Really Context Free?

- 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;

Are Programming Languages Really Context Free?

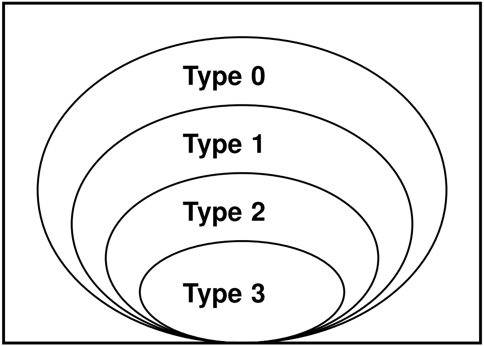
- Language from type 2 grammar
  - S $\Rightarrow$ DU $\Rightarrow$ int x; U $\Rightarrow$ int x; x=1;
  - S $\Rightarrow$ DU $\Rightarrow$ int x; U $\Rightarrow$ int x; y=1;
  - S $\Rightarrow$ DU $\Rightarrow$ int y; U $\Rightarrow$ int y; x=1;
  - S $\Rightarrow$ DU $\Rightarrow$ int y; U $\Rightarrow$ 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;

- PLs are context sensitive, why use CFG in parsing?

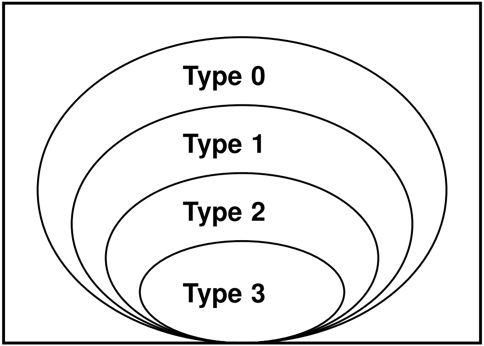
The Chomsky Hierarchy of Grammars

- RL $\subset$ CFL $\subset$ CSL $\subset$ L(Recursive Grammar)



The Chomsky Hierarchy of Grammars

- RL $\subset$ CFL $\subset$ CSL $\subset$ L(Recursive Grammar)



- However, $L_y \subset L_x$ where L_x:$[^i]^k$—RG, L_y:$[^i]^i$—CFG
  - Is it a problem?

Context Free Grammars

Syntax Analysis is a process of derivation

- Grammar is used to derive string or construct parser

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

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

Examples

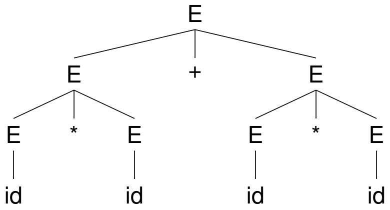
$$E \rightarrow E * E \quad | \quad E + E \quad | \quad (E) \quad | \quad id$$

- leftmost derivation
  $E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow id * E + E \Rightarrow id * id + E \Rightarrow ...$
  $\Rightarrow id * id + id * id$

- rightmost derivation
  $E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * id \Rightarrow E + id * id \Rightarrow ...$
  $\Rightarrow id * id + id * id$

A Parse Tree represents the Derivation

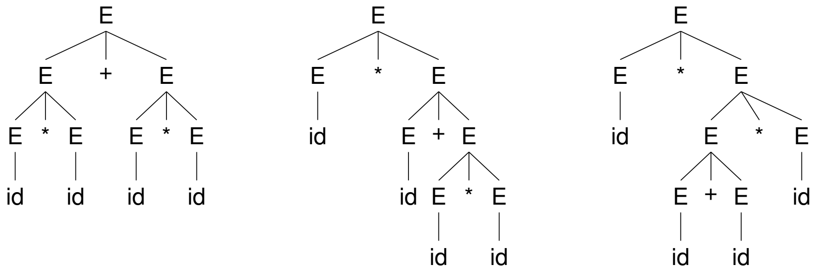
- This is the parse tree that represents both derivations:



- A parse tree
  - describes program structure (defined by the rules applied)
  - is agnostic of leftmost or rightmost derivation (as long as the same rules are applied in both)
- There are two types of nodes in a parse tree:
  - Leaves: terminals that form the sentence
  - Non-leaves: intermediate non-terminals in the derivation

Different Rules result in different Parse Trees

- Application of different rules result in different parse trees:



- Note: each parse tree has a unique leftmost derivation
  - First: E ⇒ E + E ⇒ E * E + E ⇒ id * E + E ⇒ id * id + E ⇒ ...
  - Second: E ⇒ E * E ⇒ id * E ⇒ id * E + E ⇒ id * id + E ⇒ ...
  - Third: E ⇒ E * E ⇒ id * E ⇒ id * E * E ⇒ id * E + E * E ⇒ ...
- Same goes for rightmost derivations

Ambiguity

- A grammar G is **ambiguous** if
  - there exist a string *str* $\in$ L(G) such that
  - more than one parse tree derives *str*
    $\equiv$ there is more than leftmost derivation for *str*
    $\equiv$ there is more than rightmost derivation for *str*

- Grammars that produce multiple parse trees is a problem
  - Each parse tree is a different interpretation of program

- Likely, there is an unambiguous version of the grammar
  - That accepts the same programming language
  - Programming languages are rarely inherently ambiguous

Using precendece 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

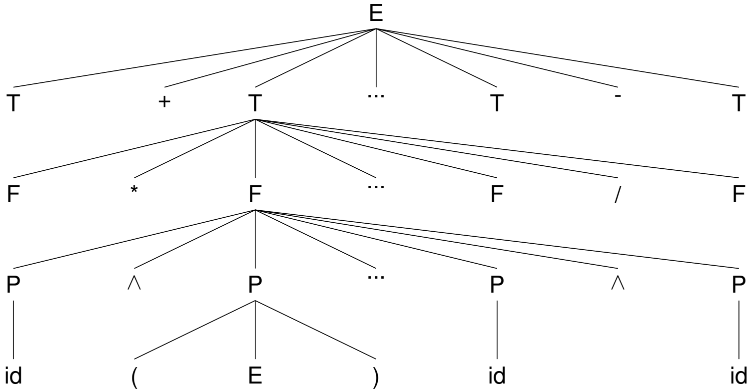
- $E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \hat{} E \mid (E) \mid id$

  rewrite it to

  $$
  \begin{aligned}
  E &\rightarrow E + T \mid E - T \mid T \\
  T &\rightarrow T * F \mid T / F \mid F \\
  F &\rightarrow P \hat{} F \mid P \\
  P &\rightarrow id \mid (E)
  \end{aligned}
  $$

Using precendece to remove ambiguity

- New grammar produces a parse tree looking like this:



- Note: some interior nodes were omitted to fit into slide.
- Levels in parse tree express levels of precedence.

Using associativity 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,

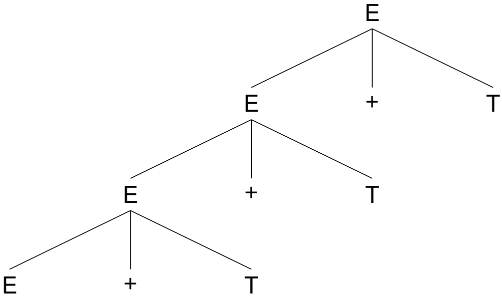
  $E \rightarrow E + E$ .... ; allows both left/right associativity

  rewrite it to

  $E \rightarrow E + T$ ... ; only left associativity
  $F \rightarrow P$ ˆ F ... ; only right associativity

Using associativity to remove ambiguity

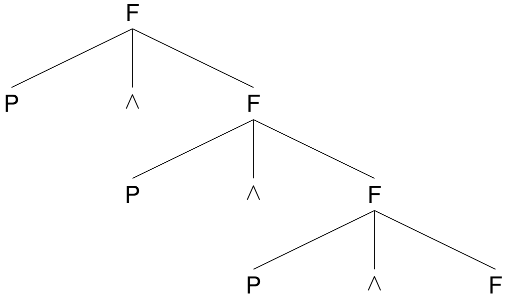
- $E \rightarrow E + T$ produces a parse tree that grows to the left:



- In this way, the parse tree expresses left associativity.
  - id + id + id = (id + id) + id

Using associativity to remove ambiguity

- $F \rightarrow P$ ˆ F produces a parse tree that grows to the right:



- In this way, the parse tree expresses right associativity.
  - id ˆ id ˆ id = id ˆ (id ˆ id)

Ambiguity is undecidable for CFGs

- Decidable: Turing Machine can answer in finite time
- It is **decidable** if a string is in a context free language
  - Implementing a parser is feasible for every CFL
- It is **decidable** if a string produces multiple parse trees
  - Only need to exhaustively generate all parse trees for string
- It is **undecidable** if a context free grammar is ambiguous
  - Checking ambiguity at compile time is impossible
  - Can only be checked reliably at runtime 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
    - Can be done easily at compile time

The Two Outcomes of Parsing

- Outcome 1: Parser is able to derive input from grammar
  - Parser builds parse tree that represents the derivation
- Outcome 2: Parser is unable to derive input from grammar
  - Parser emits a syntax error with source code location

The Two Outcomes of Parsing

- Outcome 1: Parser is able to derive input from grammar
  - Parser builds parse tree that represents the derivation
- Outcome 2: Parser is unable to derive input from grammar
  - Parser emits a syntax error with source code location
- How would you write a parser that does both well?

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 grammars
  - Starts from root and expands into leaves
  - Parser 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 wider class of grammars
  - 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
  - similar to a parse tree but ignores some details
  - internal nodes may contain terminal symbols

An Example

- Consider the grammar

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

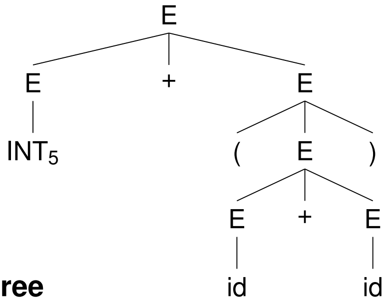
  and an input

  $$5 + (\,2 + 3\,)$$

- After lexical analysis, we have a sequence of tokens

  $$\text{INT}_5\ \text{‘+’}\ \text{‘(’}\ \text{INT}_2\ \text{‘+’}\ \text{INT}_3\ \text{‘)’}$$

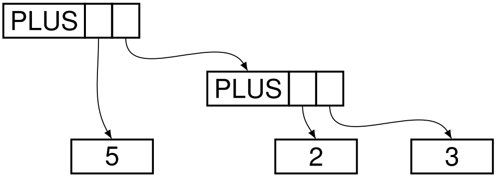
Parse Tree of the Input



- **A parse tree**
  - Traces the operation of the parser
  - Does capture the nested structure
- **but** contains too much information
  - parentheses
  - single-successor nodes

Abstract Syntax Tree

- An **Abstract Syntax Tree (AST)** for the input



  - **AST** also captures the nested structure
  - **AST** abstracts from parse tree (a.k.a. concrete syntax tree)
  - **AST** is more compact and contains only relevant info

  - **AST**s are used in most compilers rather than parse trees

How are ASTs Constructed?

- Through implementation of **semantic actions**

- We already used them in project 1 to return token tuples
- To construct AST, we attach an **attribute** to each symbol X
  - **X.ast** — the constructed AST for symbol X

- Extend each production rule with semantic actions, i.e.

$$\mathbf{X} \quad \rightarrow \quad \mathbf{Y}_1\mathbf{Y}_2\mathbf{...Y}_n \qquad \mathbf{\{\ actions\ \}}$$

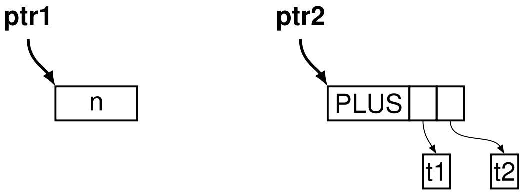
actions may define or use X.ast, Y_i.ast ($1 \leq i \leq n$)

Example

- For the previous example, we have

E	$\rightarrow$	int	{ E.ast = mkleaf(int.lval) }
	|	E1 + E2	{ E.ast = mkplus(E1.ast, E2.ast) }
	|	(E1)	{ E.ast = E1.ast }

- Here, we use two pre-defined fuctions
  - ptr1=mkleaf(n) — create a leave node and assign value “n”
  - ptr2=mkplus(t1, t2) — create a tree node and assign the root value “PLUS”, and two subtrees as t1 and t2



AST Construction Steps

For input INT_5 '+' '(' INT_2 '+' INT_3 ')'

Construction order given is for a top-down LL(1) parser

(Order can change depending on parser implementation)

AST Construction Steps

For input INT_5 '+' '(' INT_2 '+' INT_3 ')'

Construction order given is for a top-down LL(1) parser

(Order can change depending on parser implementation)

E1.ast=mkleaf(5)

$\boxed{5}$

AST Construction Steps

For input INT_5 '+' '(' INT_2 '+' INT_3 ')'

Construction order given is for a top-down LL(1) parser

(Order can change depending on parser implementation)

E1.ast=mkleaf(5) E2.ast=mkleaf(2)

$\boxed{5}$ $\boxed{2}$

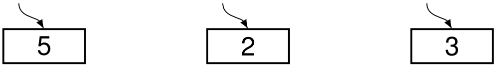
AST Construction Steps

For input INT_5 '+' '(' INT_2 '+' INT_3 ')'

Construction order given is for a top-down LL(1) parser

(Order can change depending on parser implementation)

E1.ast=mkleaf(5) E2.ast=mkleaf(2) E3.ast=mkleaf(3)

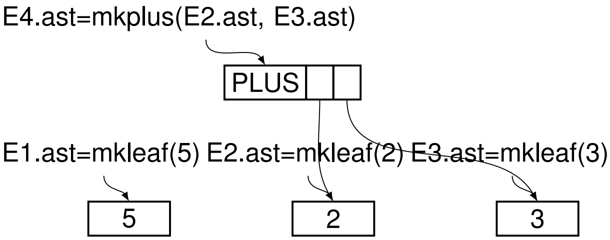


AST Construction Steps

For input INT_5 '+' '(' INT_2 '+' INT_3 ')'

Construction order given is for a top-down LL(1) parser

(Order can change depending on parser implementation)

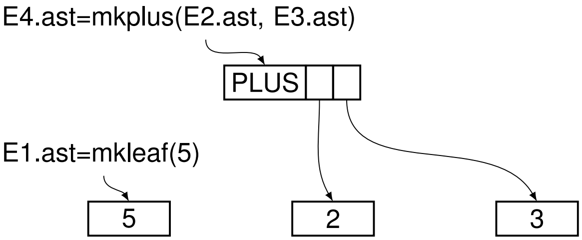


AST Construction Steps

For input INT_5 '+' '(' INT_2 '+' INT_3 ')'

Construction order given is for a top-down LL(1) parser

(Order can change depending on parser implementation)

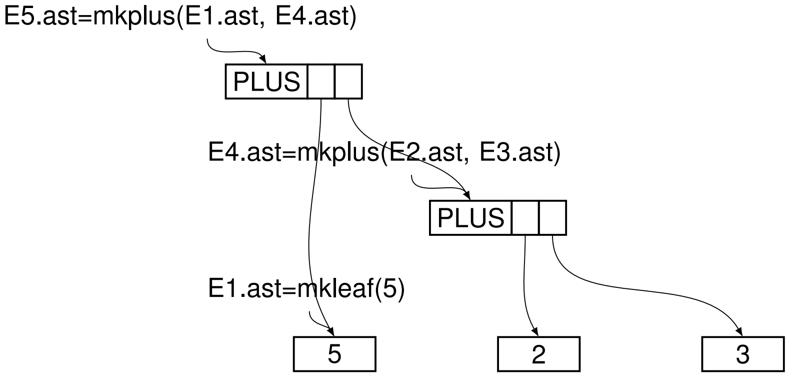


AST Construction Steps

For input INT_5 '+' '(' INT_2 '+' INT_3 ')'

Construction order given is for a top-down LL(1) parser

(Order can change depending on parser implementation)



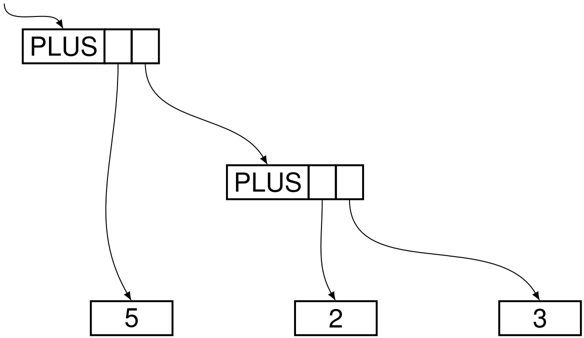
AST Construction Steps

For input INT_5 '+' '(' INT_2 '+' INT_3 ')'

Construction order given is for a top-down LL(1) parser

(Order can change depending on parser implementation)

E5.ast=mkplus(E1.ast, E4.ast)



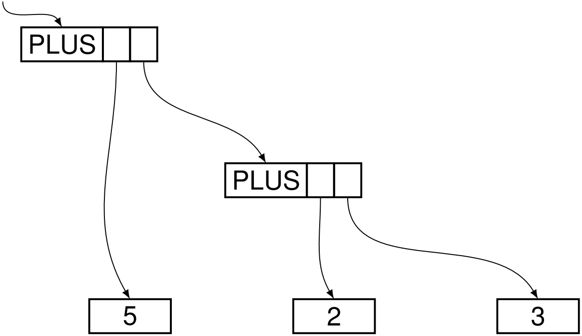
AST Construction Steps

For input INT_5 '+' '(' INT_2 '+' INT_3 ')'

Construction order given is for a top-down LL(1) parser

(Order can change depending on parser implementation)

E5.ast=mkplus(E1.ast, E4.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

Parsing

- We will study two approaches
- Top-down
  - Easier to understand and implement manually
- Bottom-up
  - More powerful, can be implemented automatically

Example

Consider a CFG grammar G

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

$D \rightarrow d \qquad C \rightarrow c$

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

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

Leftmost Derivation:

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

S

Rightmost Derivation:

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

Example

Consider a CFG grammar G

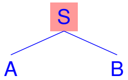
$$S \rightarrow A\,B \qquad A \rightarrow a\,C \qquad B \rightarrow b\,D$$
$$D \rightarrow d \qquad C \rightarrow c$$

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

L(G) = { acbd }

Leftmost Derivation:

S ⇒ AB (1)
⇒ aCB (2)
⇒ acB (3)
⇒ acbD (4)
⇒ acbd (5)



Rightmost Derivation:

S ⇒ AB (5)
⇒ AbD (4)
⇒ Abd (3)
⇒ aCbd (2)
⇒ acbd (1)

Example

Consider a CFG grammar G

$S \rightarrow A\,B \qquad A \rightarrow a\,C \qquad B \rightarrow b\,D$

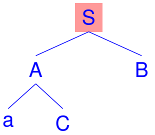
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { 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

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

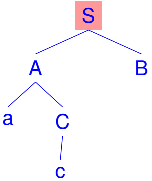
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { 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

$S \rightarrow A\,B \qquad A \rightarrow a\,C \qquad B \rightarrow b\,D$

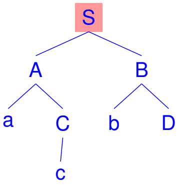
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { 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

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

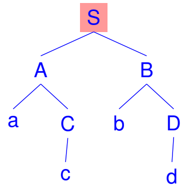
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { 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

$S \rightarrow A\,B \qquad A \rightarrow a\,C \qquad B \rightarrow b\,D$

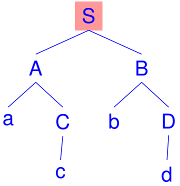
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { 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)

a c b d

Example

Consider a CFG grammar G

$S \rightarrow A\,B \qquad A \rightarrow a\,C \qquad B \rightarrow b\,D$

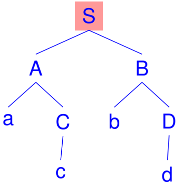
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { acbd }

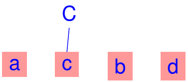
Leftmost Derivation:

S ⇒ AB (1)
⇒ aCB (2)
⇒ acB (3)
⇒ acbD (4)
⇒ acbd (5)



Rightmost Derivation:

S ⇒ AB (5)
⇒ AbD (4)
⇒ Abd (3)
⇒ aCbd (2)
⇒ acbd (1)



Example

Consider a CFG grammar G

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

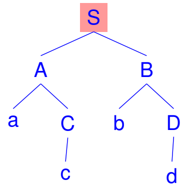
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { acbd }

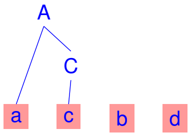
Leftmost Derivation:

$S \Rightarrow AB$ (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

$S \rightarrow A\,B \qquad A \rightarrow a\,C \qquad B \rightarrow b\,D$

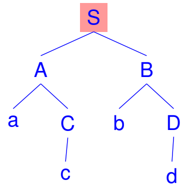
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { acbd }

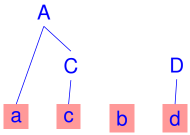
Leftmost Derivation:

S ⇒ AB (1)
⇒ aCB (2)
⇒ acB (3)
⇒ acbD (4)
⇒ acbd (5)



Rightmost Derivation:

S ⇒ AB (5)
⇒ AbD (4)
⇒ Abd (3)
⇒ aCbd (2)
⇒ acbd (1)



Example

Consider a CFG grammar G

$S \rightarrow A\,B \qquad A \rightarrow a\,C \qquad B \rightarrow b\,D$

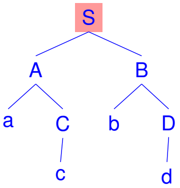
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { acbd }

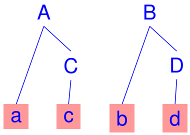
Leftmost Derivation:

$S \Rightarrow$ 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

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

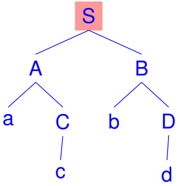
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { acbd }

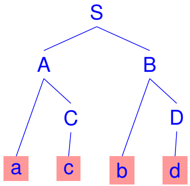
Leftmost Derivation:

$S \Rightarrow AB$ (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

Backtracking or Predictive?

- How does a parser choose between production rules?
  - Given $A \rightarrow \alpha|\beta$, expand A to α or β?
- Backtracking parser: exhaustively tries all rules
  - When input mismatch, backtrack to alternative rule
  - Con: Non-linear time due to exhaustive search
  - Con: Complex to roll back semantic actions on backtrack
  - Pro: Can parse most CFGs (except left-recursion)
- Predictive parser: predict correct rule using lookahead
  - Looks ahead k input symbols to make prediction
  - Con: Can parse only a subset of CFGs (dependent on k)
  - Pro: Linear time as only correct derivations are done
  - Pro: Simple structure as there is no need to backtrack
- Parsers can be backtracking or predictive (or both).

Recursive Descent or Table Driven?

- How is the parser implementation done?
  - Hand-coded parsers are typically recursive descent
  - Auto-generated parsers are table driven

- Recursive descent parser: each non-terminal is a function
  - Function is in charge of expanding non-terminal
  - Descends parse tree via recursive calls to non-terminals
  - Hand-written but easier to customize and control
  - Typically uses backtracking rather than prediction

- Table driven parser: uses a table of predictions
  - Similar to lexer, uses a table to decide on next production
  - Table indexed by non-terminal and *k* lookahead symbols
  - Similar to lexer, table can be generated from grammar
  - Always predictive but can use backtracking if needed

Backtracking Example

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

input string: int * int

start symbol: E

initial parse tree is E

Backtracking Example

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

input string: int * int

start symbol: E

initial parse tree is 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→T
– pick right most rule T→(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

$E \Rightarrow T \cancel{\Rightarrow (E)}$

– pick right most rule E→T
– pick right most rule T→(E)
– "(" does not match "int"
– failure, backtrack one level

Parsing Sequence using Backtracking

$E \Rightarrow T \cancel{\Rightarrow (E)}$

$\Rightarrow$ int

– pick right most rule E→T
– pick right most rule T→(E)
– “(” does not match “int”
– failure, backtrack one level
– pick T→int
– “int” matches input “int”

Parsing Sequence using Backtracking

$E \;\Rightarrow\; T \;\cancel{\Rightarrow\; (\,E\,)}$

$\qquad\qquad \Rightarrow\; \text{int}$

- pick right most rule E→T
- pick right most rule T→(E)
- "(" does not match "int"
- failure, backtrack one level
- pick T→int
- "int" matches input "int"
- however, we expect more tokens
- failure, backtrack one level

Parsing Sequence using Backtracking

$E \Rightarrow T \cancel{\Rightarrow (E)}$

$\cancel{\Rightarrow \text{int}}$

- pick right most rule E→T
- pick right most rule T→(E)
- "(" does not match "int"
- failure, backtrack one level
- pick T→int
- "int" matches input "int"
- however, we expect more tokens
- failure, backtrack one level

Parsing Sequence using Backtracking

$E \Rightarrow T \cancel{\Rightarrow (E)}$	– pick right most rule E$\rightarrow$T
	– pick right most rule T$\rightarrow$(E)
	– "(" does not match "int"
	– failure, backtrack one level
$\cancel{\Rightarrow \text{int}}$	– pick T$\rightarrow$int
	– "int" matches input "int"
	– however, we expect more tokens
	– failure, backtrack one level
$\Rightarrow$ int * T	– pick T$\rightarrow$int * T

Parsing Sequence using Backtracking

$E \Rightarrow T \cancel{\Rightarrow (E)}$	– pick right most rule E→T
	– pick right most rule T→(E)
	– “(” does not match “int”
	– failure, backtrack one level
$\cancel{\Rightarrow \text{int}}$	– pick T→int
	– “int” matches input “int”
	– however, we expect more tokens
	– failure, backtrack one level
$\Rightarrow \text{int} * T \Rightarrow \text{int} * (E)$	– pick T→int * T
	– pick T→int * (E)

Parsing Sequence using Backtracking

$E \Rightarrow T \cancel{\Rightarrow (E)}$	– pick right most rule E→T
	– pick right most rule T→(E)
	– "(" does not match "int"
	– failure, backtrack one level
$\cancel{\Rightarrow \text{int}}$	– pick T→int
	– "int" matches input "int"
	– however, we expect more tokens
	– failure, backtrack one level
$\Rightarrow \text{int * T} \Rightarrow \text{int * (E)}$	– pick T→int * T
	– pick T→int * (E)
	– "(" does not match input "int"
	– failure, backtrack one level

Parsing Sequence using Backtracking

E ⇒ T ~~⇒ (E)~~	– pick right most rule E→T
	– pick right most rule T→(E)
	– "(" does not match "int"
	– failure, backtrack one level
~~⇒ int~~	– pick T→int
	– "int" matches input "int"
	– however, we expect more tokens
	– failure, backtrack one level
⇒ int * T ~~⇒ int * (E)~~	– pick T→int * T
	– pick T→int * (E)
	– "(" does not match input "int"
	– failure, backtrack one level

Parsing Sequence using Backtracking

$E \Rightarrow T$ ~~$\Rightarrow$ (E)~~	– pick right most rule E→T
	– pick right most rule T→(E)
	– "(" does not match "int"
	– failure, backtrack one level
~~$\Rightarrow$ int~~	– pick T→int
	– "int" matches input "int"
	– however, we expect more tokens
	– failure, backtrack one level
$\Rightarrow$ int * T ~~$\Rightarrow$ int * (E)~~	– pick T→int * T
	– pick T→int * (E)
	– "(" does not match input "int"
	– failure, backtrack one level
$\Rightarrow$ int * int	– pick T→int

Parsing Sequence using Backtracking

$E \Rightarrow T \cancel{\Rightarrow (E)}$	– pick right most rule E→T
	– pick right most rule T→(E)
	– "(" does not match "int"
	– failure, backtrack one level
$\cancel{\Rightarrow \text{int}}$	– pick T→int
	– "int" matches input "int"
	– however, we expect more tokens
	– failure, backtrack one level
$\Rightarrow \text{int} * T \cancel{\Rightarrow \text{int} * (E)}$	– pick T→int * T
	– pick T→int * (E)
	– "(" does not match input "int"
	– failure, backtrack one level
$\Rightarrow \text{int} * \text{int}$	– pick T→int
	– match, **accept**

Recursive Descent Parser with Backtracking

Recursive Descent Parsing Implementation

- When expanding a non-terminal, try all productions 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

- Create a function 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 function for that non-terminal
     c. If match succeeds for entire RHS, return success
     d. If match fails, regurgitate input and try next RHS
  2. If match succeeds for any rule, apply that rule to LHS

- If entire input string matched with start symbol, success!

A Hand-coded Recursive Descent Parser

- Sample implementation of parser for previous grammar:

$$\begin{array}{lllllll} E & \rightarrow & T + E & \mid & T & & \\ T & \rightarrow & \text{int} * T & \mid & \text{int} & \mid & (E) \end{array}$$

```
char fetchNext() {
   // Fetch one character
}
void regurgitate(int n) {
   // Unfetch n characters
}
bool expr() {
rule1:
    if(!term()) goto rule2;
    if (fetchNext()!=AddNum) {
      regurgitate(1);
      goto rule2;
    }
    if(!expr()) {
      regurgitate(1);
      goto rule2;
    }
    return true;
rule2:
    if (!term()) return false;
    return true;
}
```

```
bool term() {
rule1:
    if (fetchNext()!=IntNum) {
      regurgitate(1);
      goto rule2;
    }
    if (fetchNext()!=StarNum) {
      regurgitate(2);
      goto rule2;
    }
    if(!term()) {
      regurgitate(2);
      goto rule2;
    }
    return true;
rule2:
    ...
    return true;
rule3:
    ...
    return true;
}
```

Recursive Descent has a Left Recursion Problem

- Recursive descent doesn't work if grammar is left recursive

- 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?

Recursive Descent has a Left Recursion Problem

- Recursive descent doesn't work if grammar is left recursive
- 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

Removing Left Recursion

- All immediate left recursion can be eliminated this way:

  $A \rightarrow A\ x \mid y$

  change to

  $A \rightarrow y\ A'$

  $A' \rightarrow x\ A' \mid \varepsilon$

- Not all left recursion is immediate

  (Recursion may involve multiple non-terminals)

  $A \rightarrow BC \mid D$

  $B \rightarrow AE \mid F$

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

  ... (not required for this course)

Table Driven Parser using Predictions

Predictive Parsers can avoid Backtracking

- Predict correct production rule based on *k* lookahead
  - Backtracking can be avoided if grammar limited to LL(k)
- 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 parse-able by 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)!

Left factoring can make grammars LL(1)

- An LL(1) grammar
  - First terminal of every alternative production is unique

    $A \rightarrow a\ B\ D \mid b\ B\ B$

    $B \rightarrow c \mid b\ c\ e$

    $D \rightarrow d$
- What if no LL(1)? Left factor to make it LL(1)!
  - What if production rules for A was changed to below?

    $A \rightarrow a\ B\ D \mid a\ B\ B$
  - Left factor $a\ B$ to enable prediction:

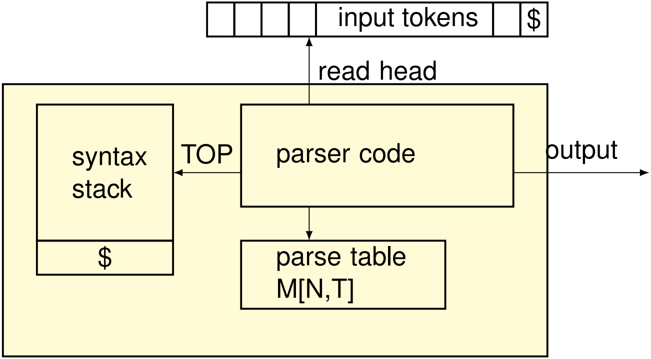
    $A \rightarrow a\ B\ A' \mid a\ B\ A'$

    $A' \rightarrow D \mid B$
- In general, if you see $A \rightarrow \alpha\beta \mid \alpha\gamma$, change to:

  $A \rightarrow \alpha\ A'$

  $A' \rightarrow \beta \mid \gamma$

A Table Driven Pushdown Automaton



Syntax stack — hold right hand side (RHS) of grammar rules
Parse table M[A,b] — an entry containing rule "A $\rightarrow$ ..." or error
Parser code — next action based on **(current token, stack top)**
Table can be automatically generated from grammar (just like lexers)

A Sample Parse Table

	int	*	+	(	)	$
E	$E \rightarrow TX$			$E \rightarrow TX$		
X			$X \rightarrow +E$		$X \rightarrow \varepsilon$	$X \rightarrow \varepsilon$
T	$T \rightarrow int\ Y$			$T \rightarrow (E)$		
Y		$Y \rightarrow *T$	$Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$

- Predicts rule based on (current non-terminal, lookahead)
  - **First column** lists all non-terminals
  - **First row** lists all possible terminals and $
  - A table entry contains one production (one prediction)
- What if an entry has more than one production?
  - Means that this grammar is not LL(1)
  - A parser can handle this situation by either:
    - Throwing an error to grammar writer to fix the problem
    - Resorting to backtracking to try out both productions

Pseudocode for Table-Driven Parser

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 **and** advance input head
  - If X!=a, then
    Case (a): if X $\in$ T, then
    - parser halts with "failed", input rejected

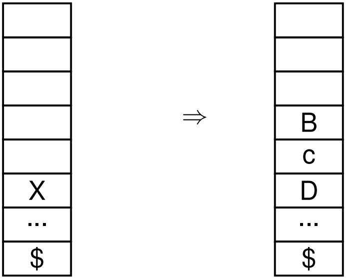
    Case (b): if X $\in$ N, M[X,a] = "X$\rightarrow$RHS"
    - pop X **and** push RHS to stack in reverse order

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”



- Why? Because that is the order of leftmost derivation.

Applying LL(1) Parsing to a Grammar

- Given our old grammar

$$\begin{array}{lll} E & \rightarrow & T + E \mid T \\ T & \rightarrow & \text{int} * T \mid \text{int} \mid (E) \end{array}$$

  - Requires left factoring of T and int

- After rewriting grammar, we have

$$\begin{array}{lll} E & \rightarrow & T\,X \\ X & \rightarrow & + E \mid \varepsilon \\ T & \rightarrow & \text{int}\;Y \mid (E) \\ Y & \rightarrow & * T \mid \varepsilon \end{array}$$

Using the Parse Table

- To recognize “int * int”

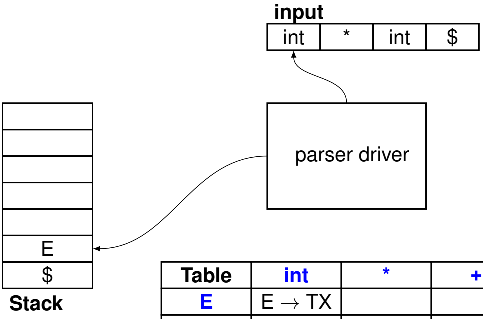


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

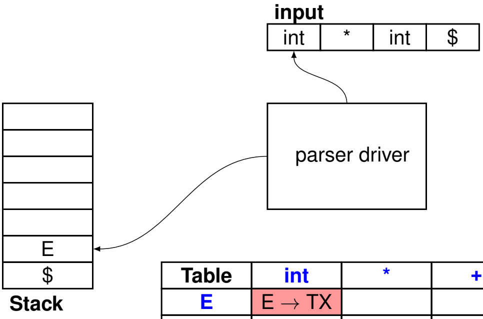


Table	int	*	+	(	)	$
E	$E \rightarrow TX$			$E \rightarrow TX$		
X			$X \rightarrow +E$		$X \rightarrow \varepsilon$	$X \rightarrow \varepsilon$
T	$T \rightarrow$ int Y			$T \rightarrow (E)$		
Y		$Y \rightarrow *T$	$Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

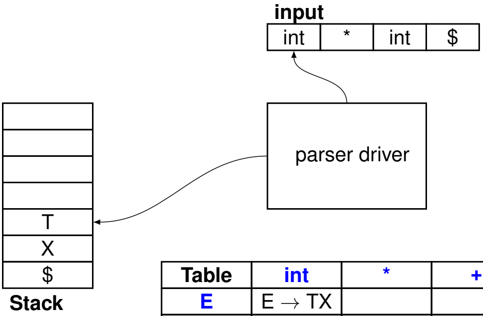


Table	int	*	+	(	)	$
E	$E \to TX$			$E \to TX$		
X			$X \to +E$		$X \to \varepsilon$	$X \to \varepsilon$
T	$T \to \text{int } Y$			$T \to (E)$		
Y		$Y \to *T$	$Y \to \varepsilon$		$Y \to \varepsilon$	$Y \to \varepsilon$

Using the Parse Table

- To recognize “int * int”

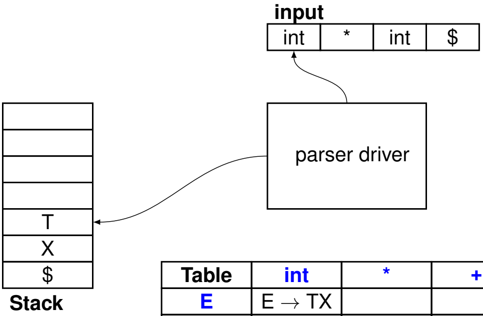


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize "int * int"

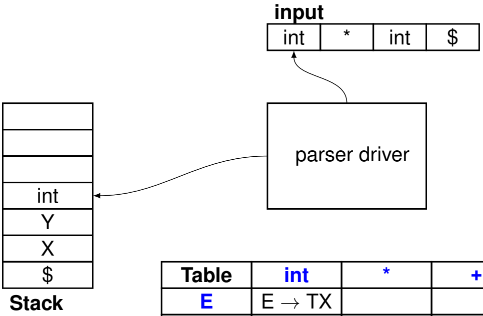


Table	int	*	+	(	)	$
E	$E \rightarrow TX$			$E \rightarrow TX$		
X			$X \rightarrow +E$		$X \rightarrow \varepsilon$	$X \rightarrow \varepsilon$
T	$T \rightarrow int\ Y$			$T \rightarrow (E)$		
Y		$Y \rightarrow *T$	$Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

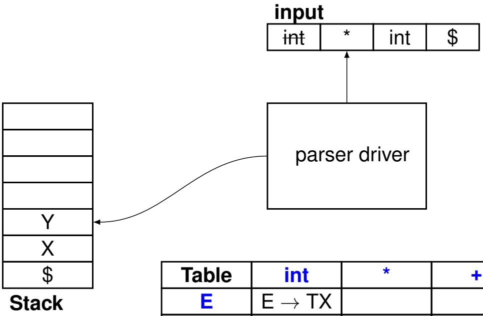


Table	int	*	+	(	)	$
E	$E \rightarrow TX$			$E \rightarrow TX$		
X			$X \rightarrow +E$		$X \rightarrow \varepsilon$	$X \rightarrow \varepsilon$
T	$T \rightarrow$ int Y			$T \rightarrow (E)$		
Y		$Y \rightarrow *T$	$Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

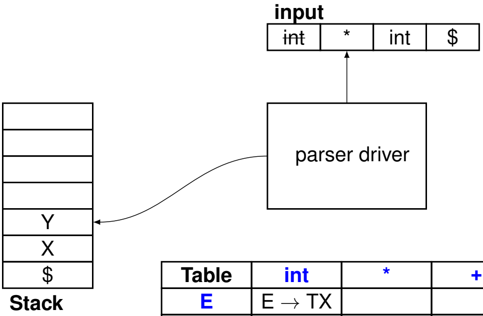


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

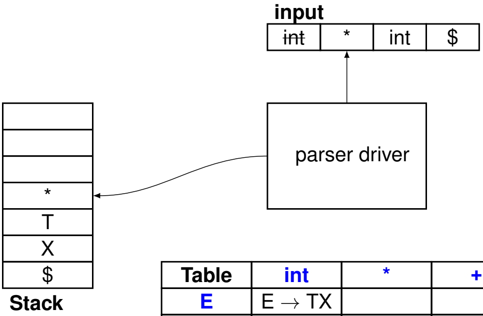


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

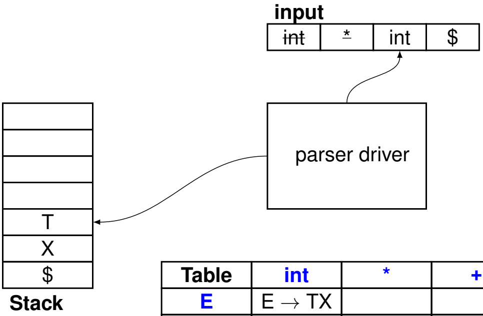


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow$ ε	X $\rightarrow$ ε
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow$ ε		Y $\rightarrow$ ε	Y $\rightarrow$ ε

Using the Parse Table

- To recognize “int * int”

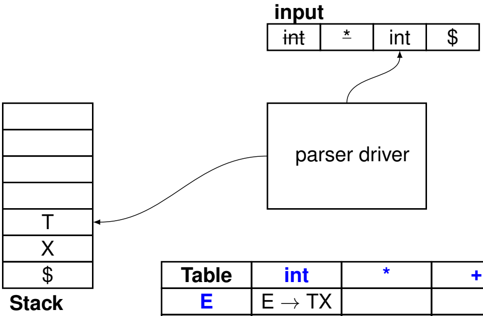


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

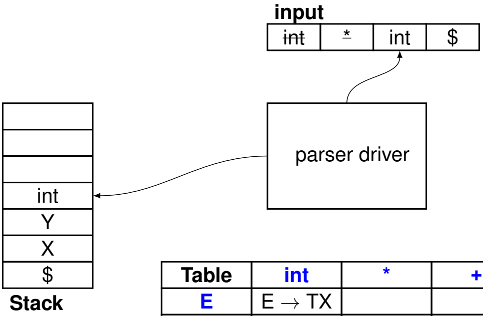
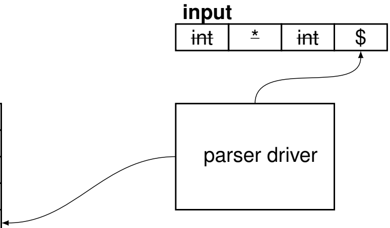


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”



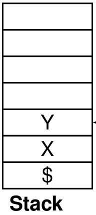


Table	**int**	*****	**+**	**(**	**)**	**$**
E	$E \to TX$			$E \to TX$		
X			$X \to +E$		$X \to \varepsilon$	$X \to \varepsilon$
T	$T \to$ int Y			$T \to (E)$		
Y		$Y \to *T$	$Y \to \varepsilon$		$Y \to \varepsilon$	$Y \to \varepsilon$

Using the Parse Table

- To recognize "int * int"

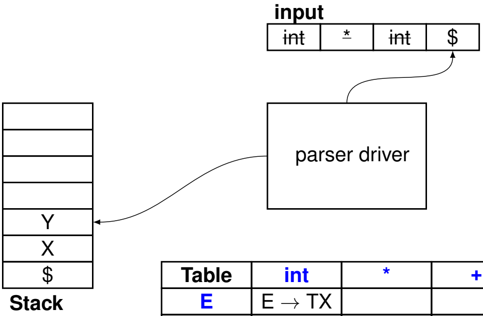


Table	int	*	+	(	)	$
E	$E \rightarrow TX$			$E \rightarrow TX$		
X			$X \rightarrow +E$		$X \rightarrow \varepsilon$	$X \rightarrow \varepsilon$
T	$T \rightarrow$ int Y			$T \rightarrow (E)$		
Y		$Y \rightarrow *T$	$Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

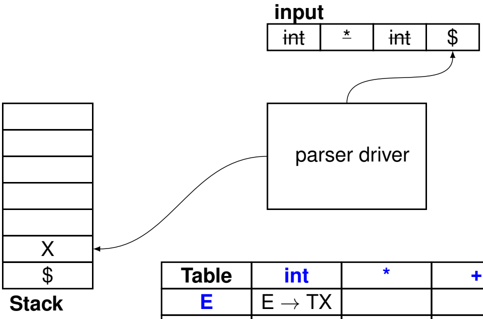


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

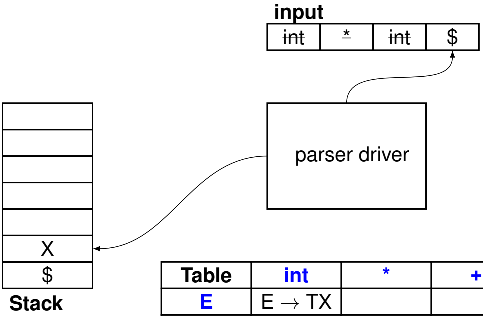


Table	int	*	+	(	)	$
E	$E \rightarrow TX$			$E \rightarrow TX$		
X			$X \rightarrow +E$		$X \rightarrow \varepsilon$	$X \rightarrow \varepsilon$
T	$T \rightarrow$ int Y			$T \rightarrow (E)$		
Y		$Y \rightarrow *T$	$Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

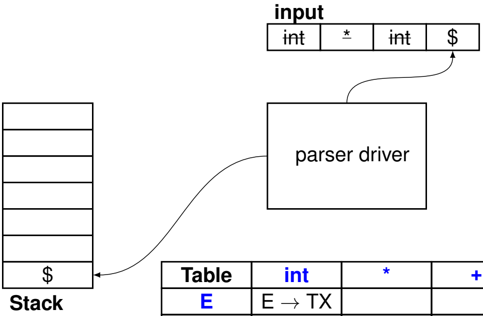
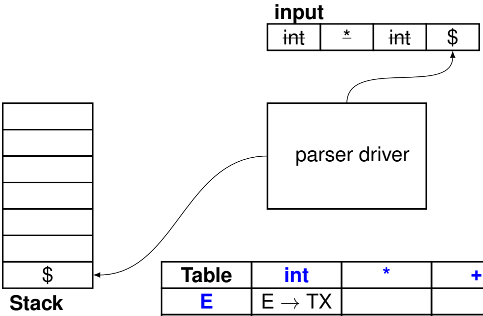


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”



Accept!!!

Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Recognition Sequence

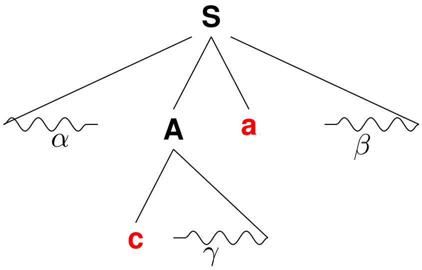
- It is possible to write in a action list

Stack	Input	Action
E $	int * int $	E→ TX
T X $	int * int $	T→ int Y
int Y X $	int * int $	terminal
Y X $	* int $	Y→ * T
* T X $	* int $	terminal
T X $	int $	T→ int Y
int Y X $	int $	terminal
Y X $	$	Y→ ε
X $	$	X→ ε
$	$	halt and accept

First step in building Parse Table: First and Follow Sets

- First(α) = $\{t | \alpha \Rightarrow * t\beta\}$
  - Set of terminals that can start a string derived from α.
- Follow(α) = $\{t | S \Rightarrow * \alpha t \beta\}$
  - Set of terminals that can follow α in some derivation.
- Given rule $A \rightarrow \alpha$,
  - Choose $A \rightarrow \alpha$ for all terminals in First(α)
  - Choose $A \rightarrow \alpha$ for all terminals in Follow(A), if and only if $\alpha \Rightarrow * \varepsilon$

Intuitive Meaning of **First** and **Follow**



c $\in$ **First(A)**

a $\in$ **Follow(A)**

- Why is the Follow Set important?

Calculating First(α)

- Given $A \rightarrow \alpha$, let's calculate First(α).
  - α is string $Y_1 Y_2 Y_3 ... Y_m$ of terminals and non-terminals.
  1) For all Y_i, if Y_i is a terminal t, then First(Y_i) = t
  2) For all non-terminal Y_i, recursively calculate First(Y_i)
     (If $Y_i \rightarrow \beta \mid \gamma$, First($Y_i$) = First($\beta$) $\cup$ First(γ))
  3) Calculate First(α) based on First(Y_i) where $i = 1...m$

- Apply following rules until no terminal or ε can be added
  1) Add (First(Y_1) - ε) to First(α).
  2) If First(Y_1), ..., First(Y_{k-1}) all contain ε,
     then add ($\sum_{1 \leq i \leq k} First(Y_i) - \varepsilon$) to First($\alpha$).
  3) If First(Y_1), ..., First(Y_m) all contain ε,
     then add ε to First(α).

Calculating Follow(A)

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

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

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

- Apply following rules until no terminal or ε can be added

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

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

  3). Look at N on the RHS that is not followed by anything,

  if ($X \rightarrow \alpha A$) or ($X \rightarrow \alpha A \beta$ and $\varepsilon \in$ First(β)),
  then Follow(X) $\subseteq$ Follow(A)

Calculating First and Follow Sets for the example

$$
\begin{array}{lcl}
E & \rightarrow & T\ X \\
X & \rightarrow & +\ E \ \mid\ \varepsilon \\
T & \rightarrow & \text{int}\ Y \ \mid\ (\ E\ ) \\
Y & \rightarrow & *\ T \ \mid\ \varepsilon
\end{array}
$$

- Start by calculating the First Sets for all RHSs
  - First(T X)
  - First(+ E), First(ε)
  - First(int Y), First((E))
  - First(* T), First(ε)

- If any of the above First Sets contains ε, calculate the Follow Set for corresponding non-terminal

Calculating First and Follow Sets for the example

$$
\begin{array}{lcl}
E & \rightarrow & T\ X \\
X & \rightarrow & +\ E \mid \varepsilon \\
T & \rightarrow & \text{int}\ Y \mid (\ E\ ) \\
Y & \rightarrow & *\ T \mid \varepsilon
\end{array}
$$

Symbol	First
(	(
)	)
+	+
*	*
int	int
E	(, int
X	+, ε
T	(, int
Y	*, ε

RHS	First
T X	(,int
+ E	+
ε	ε
int Y	int
(E)	(
* T	*
ε	ε

Calculating First and Follow Sets for the example

$$
\begin{array}{lcl}
E & \rightarrow & T\ X \\
X & \rightarrow & +\ E \mid \varepsilon \\
T & \rightarrow & \text{int}\ Y \mid (\ E\ ) \\
Y & \rightarrow & *\ T \mid \varepsilon
\end{array}
$$

Symbol	First
(	(
)	)
+	+
*	*
int	int
E	(, int
X	+, ε
T	(, int
Y	*, ε

RHS	First
T X	(,int
+ E	+
ε	ε
int Y	int
(E)	(
* T	*
ε	ε

Non-terminal	Follow
X	$,)
Y	$,),+
E	$,)
T	$,),+

Construction of LL(1) Parse Table

- To construct the parse table, we check each A$\rightarrow$ α
  - For each terminal $\mathtt{a}$ $\in$ First(α), then add A$\rightarrow$ α to M[A,$\mathtt{a}$].
  - If ε $\in$ First(α), then
    for each terminal $\mathtt{b}$ $\in$ Follow(A), add A$\rightarrow$ α to M[A,$\mathtt{b}$].
  - If ε $\in$ First(α) and \$ $\in$ Follow(A), then add A$\rightarrow$ α to M[A,\$].

Example

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

RHS	First
T X	(,int
+ E	+
ε	ε
int Y	int
(E)	(
* T	*
ε	ε

Non-terminal	Follow
X	$,)
Y	$,),+

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

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 $\rightarrow \alpha|\beta$

  - First(α) $\cap$ First(β) = ϕ
  - at most one of α and β can derive ε
  - If β derives ε, then First(α) $\cap$ Follow(A) = ϕ

Left-recursion disqualifies grammar for LL(1)

- Recall recursive descent had trouble with left-recursion.
- Table-driven parsers have a similar problem.

- Left-recursion is of the form $A \rightarrow Ab|a$ or $A \rightarrow Ab|\varepsilon$
  - For $A \rightarrow Ab|a$, First(Ab) $\cap$ First(a) = $\{a\}$
  - For $A \rightarrow Ab|\varepsilon$, First($Ab$) $\cap$ Follow(A) = $\{b\}$
  - Either way, an ambiguity in prediction

- Even if prediction can be made with more lookahead,
  - Sentence can grow indefinitely w/o consuming input
  - We may repeatedly choose to apply $A \rightarrow Ab$:

    $A \Rightarrow A\ b \Rightarrow A\ b\ b$ ...
  - Same stack explosion problem as with recursive descent

Dealing with Non-LL(1) Grammars

(1) Likely still an LL(1) language. Massage to LL(1) grammar:
- Apply left-factoring
- Apply left-recursion removal

(2) If (1) fails, the possibilities are...
- Grammar just needs a little more lookahead
  (May need LL(k) parser where k > 1 or backtracking)
- Grammar is ambiguous (multiple parse trees)

- How do we deal with ambiguous grammars then?
  - Note: left-factoring and left-recursion removal don't help
  - Expressing precedence and associativity in grammar helps

Ambiguous not just non-LL(1)

- Some grammars are not LL(1) even after left-factoring and left-recursion removal

  S $\rightarrow$ if C then S | if C then S else S | a (other statements)
  C $\rightarrow$ b

  change to

  S $\rightarrow$ if C then S X | a
  X $\rightarrow$ else S | ε
  C $\rightarrow$ b

  problem sentence: "if b then if b then a else a"

  First(X) = {else, ε}
  From S $\rightarrow$ if C then S X , Follow(S) $\subseteq$ Follow(X)
  Follow(X) = {else, $}
  For X $\rightarrow$ else S | ε, First(else S) $\cap$ Follow(X) = {else}

- Such grammars are potentially ambiguous

Removing Ambiguity

- We want to express precedence of if-then-else over if-then.
- How would you rewrite grammar to express precedence?

  S $\rightarrow$ if C then S | S2
  S2 $\rightarrow$ if C then S2 else S | a
  C $\rightarrow$ b
- Now grammar is unambiguous but it is not LL(k) for any k
  - Intuitively, must lookahead until 'else' to choose rule for 'S'
  - That lookahead may be an arbitrary number of tokens
- Changing the grammar to be perfectly unambiguous
  - Can be very taxing for programmers to specify correctly
  - May still result in grammar not suitable for LL(1) parsing
- More practical to encode precedence rules into parser
  - E.g. Always choose X $\rightarrow$ else S over X $\rightarrow$ ε on 'else' token

LL(1) Time and Space Complexity

- LL(1) parsers operate in linear time and space relative to the length of input.
- Time: each token is processed constant number of times
  - Why?
- Space: stack space required is at max the length of input
  - If X $\rightarrow \varepsilon$ rules removed (easily done by substitution)
  - Why?
- How about LL(k)?
  - Same time complexity as the same argument applies
  - Space complexity is $O(T^k)$, where T is number of terminals (if constructing the parse table naively)

ANTLR: A modern LL(*) parser

- A free open source top-down LL(*) parser (antlr.org)
  - LL(*): can use arbitrary lookahead to parse grammar
  - Used in Apache Groovy, Jython, MySQL Workbench, ...
- Reduces table space by expressing lookahead as DFA
  - A DFA decides on which rule for each non-terminal
  - DFA can express arbitrarily long lookahead compactly
  - If DFA fails prediction, fall back to backtracking
- To learn more, refer to this paper:

  LL(*): The Foundation of the ANTLR Parser Generator

  https://www.antlr.org/papers/LL-star-PLDI11.pdf

ANTLR: A modern LL(k) parser

- Given following grammar requiring arbitrary lookahead:

  S → ID
  | ID = expr
  | (unsigned)* int ID
  | (unsigned)* ID ID

- ANTLR constructs below DFA for non-terminal S: