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?

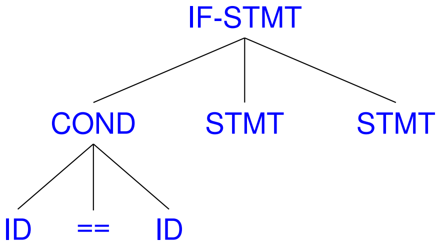
- **Parse tree**: a tree that represents grammatical structure
- Language constructs often have recursive structures

  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

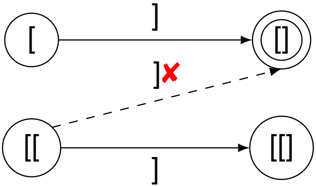
- Can regular expressions express this construct?
  - ➢ Recall RL $\equiv$ L(Regular Expression) $\equiv$ L(Finite Automata)
  - ➢ Boils down to whether an FA can accept this construct

RE/FA is Not Powerful Enough

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

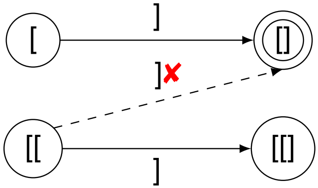
RE/FA is Not Powerful Enough

- Describe strings with pattern $[^i]^i$ **(i≥1)**
  - "[", "[]" are different states as only the latter is accepting
  - "[", "[[" are different states as only the former accepts on "]"



RE/FA is Not Powerful Enough

- Describe strings with pattern $[^i]^i$ **(i≥1)**
  - "[", "[]" are different states as only the latter is accepting
  - "[", "[[" are different states as only the 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

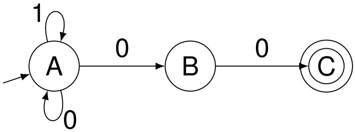
- REs cannot express recursive language constructs
- Programming languages belong to a category called CFLs
  - CLF is short for Context Free Language
  - CFLs are a strictly larger set than RLs
- To express CFLs, we need a new formalism: Grammars
- Grammars are general enough to express most languages
  - 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 $\rightarrow \varepsilon$; 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

    $$A \rightarrow \gamma$$

    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 = $\{ [^i]^j \;\;|\;\; i, j >= 1\}$
  - Regular grammar

    $S \rightarrow [\, S \mid [\, T$
    $T \rightarrow]\, T \mid]$

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

    $S \rightarrow [\, S\,] \mid [\,]$

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?

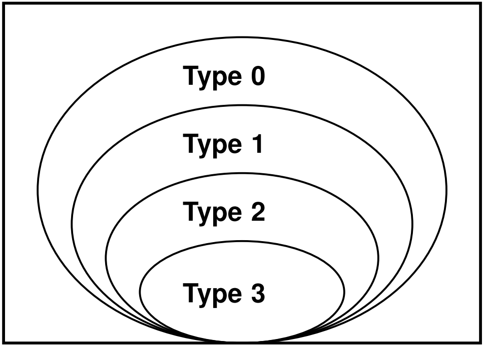
- Language from type 2 grammar
  - $S \Rightarrow DU \Rightarrow$ int x; $U \Rightarrow$ int x; x=1;
  - $S \Rightarrow DU \Rightarrow$ int x; $U \Rightarrow$ int x; y=1;
  - $S \Rightarrow DU \Rightarrow$ int y; $U \Rightarrow$ int y; x=1;
  - $S \Rightarrow DU \Rightarrow$ int y; $U \Rightarrow$ int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

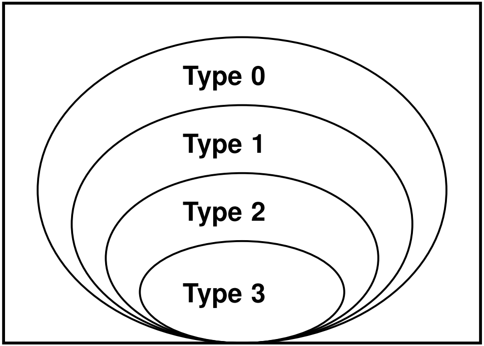
The Chomsky Hierarchy of Grammars

- 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 ⇒ ... ⇒ ... ⇒ ... ⇒ (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

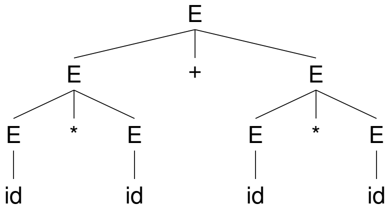
$$E \rightarrow E * E \mid E + E \mid (E) \mid id$$

- 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 represent the Derivation

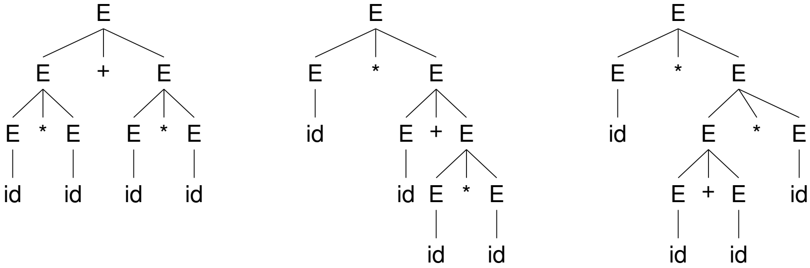
- This is the parse tree that represents both derivations:



- 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

Grammar can be rewritten 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

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

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 .... ; allows both left/right associativity

  rewrite it to

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

Ambiguity is undecidable for CFGs

- Decidable: computable using a Turing Machine
- It is **decidable** if a string is in a context free language
  - Implementing a parser is feasible for every CFL
- It is **undecidable** if a CFG 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

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

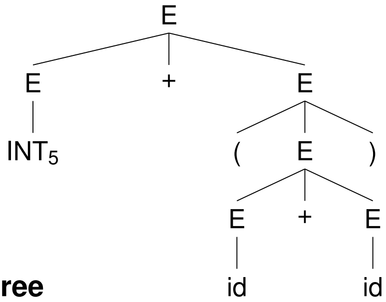
  and an input

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

- After lexical analysis, we have a sequence of tokens

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

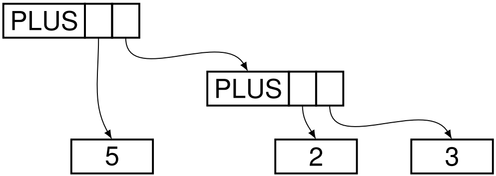
Parse Tree of the Input



- **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.

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

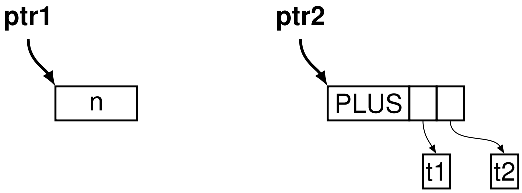
actions may define or use X.ast, Y_i.ast ($1 \leq i \leq n$)

Example

- For the previous example, we have

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

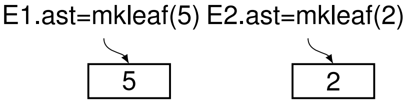
$\boxed{5}$

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)

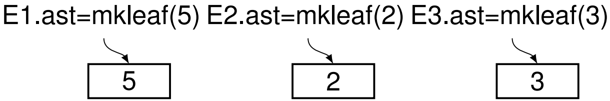


AST Construction Steps

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)

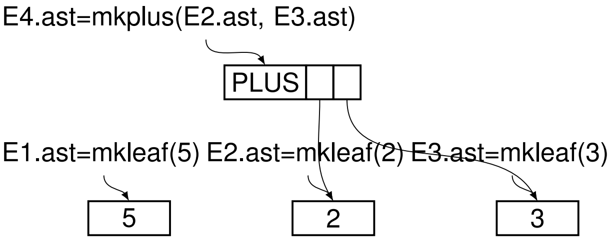


AST Construction Steps

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)

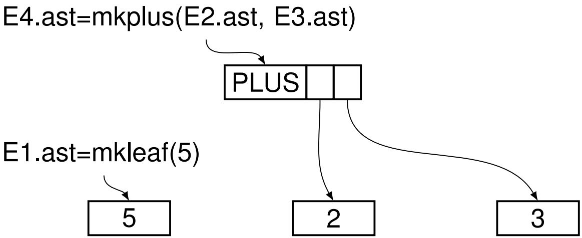


AST Construction Steps

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)

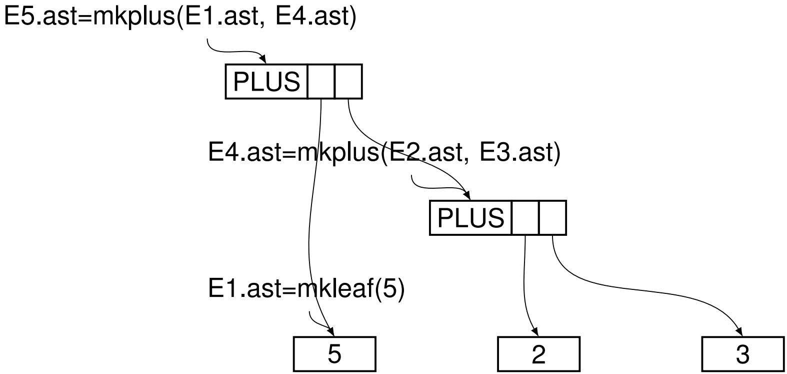


AST Construction Steps

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)



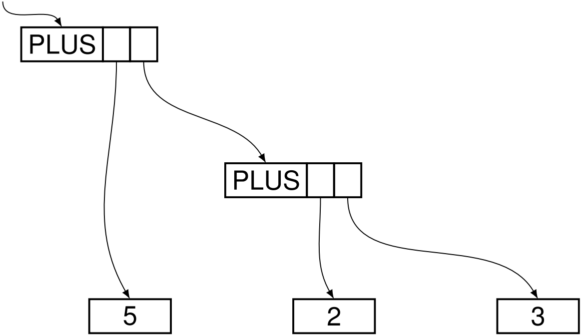
AST Construction Steps

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)



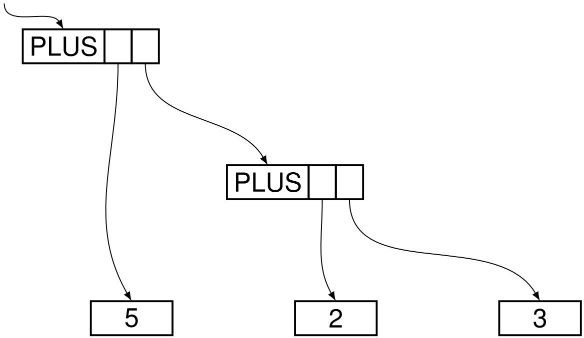
AST Construction Steps

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

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

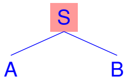
$D \to d \qquad C \to c$

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

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

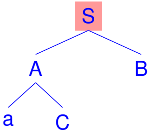
$D \rightarrow d \qquad C \rightarrow c$

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

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

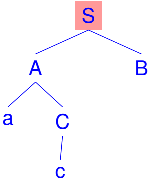
$D \rightarrow d \qquad C \rightarrow c$

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

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

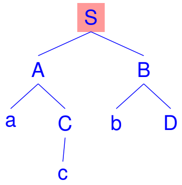
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { acbd }

Leftmost Derivation:

S ⇒ 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

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

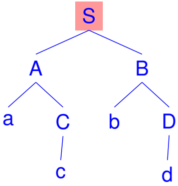
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { acbd }

Leftmost Derivation:

S ⇒ 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

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

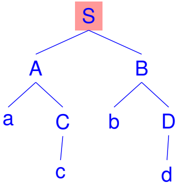
$D \rightarrow d \qquad C \rightarrow c$

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)

a c b d

Example

Consider a CFG grammar G

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

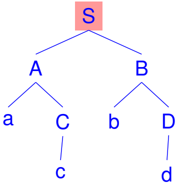
$D \rightarrow d \qquad C \rightarrow c$

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

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

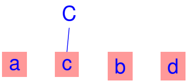
Leftmost Derivation:

$S \Rightarrow AB$ (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

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

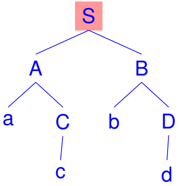
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { acbd }

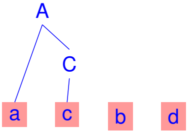
Leftmost Derivation:

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

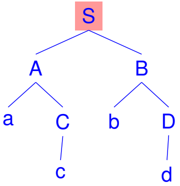
$$S \rightarrow A\,B \qquad A \rightarrow a\,C \qquad B \rightarrow b\,D$$
$$D \rightarrow d \qquad C \rightarrow c$$

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

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

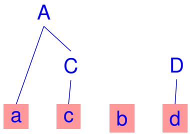
Leftmost Derivation:

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

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

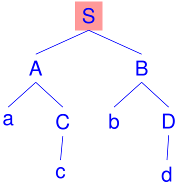
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { acbd }

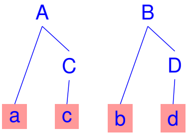
Leftmost Derivation:

$S \Rightarrow AB$ (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

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

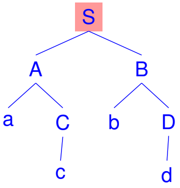
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { acbd }

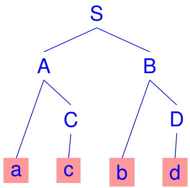
Leftmost Derivation:

$S \Rightarrow AB$ (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

- Recursive descent parser
  - Implemented using recursive calls to functions that implement the expansion of each non-terminal
  - Simple to implement, use backtracking on mismatch
- 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

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

input string: int * int

start symbol: E

initial parse tree is E

Recursive Descent Example

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

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→T
– pick right most rule T→(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 $E \to T$
– pick right most rule $T \to (E)$
– “(” 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)

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

$\Rightarrow$ 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"
– 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→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
$\Rightarrow \text{int} * T$	– pick T→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)

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

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 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
- Parsing fails if no production for the start symbol generates the entire input
- Terminals of the derivation are compared against input
  - Match — advance input, continue parsing
  - Mismatch — backtrack, or fail

Recursive Descent Parsing Algorithm

- 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

A Hand-coded Recursive Descent Parser

- Sample implementation of parser for previous grammar:

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

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

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

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
    $A \Rightarrow A\ b \Rightarrow A\ b\ b\ ...$
  - 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)

Summary of Recursive Descent

- 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 makes compilation very inefficient
  - 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

Predictive Parsers do no Backtracking

- Predict production of non-terminal based on input
  - Limit grammar so 100% prediction accuracy is possible, so that backtracking can be avoided completely!
  - First terminal of every alternative production is unique

    A $\rightarrow$ a B D | b B B
    B $\rightarrow$ c | b c e
    D $\rightarrow$ d

    parsing an input "abced" has no backtracking
  - Left factoring to enable prediction

    A $\rightarrow \alpha\beta \mid \alpha\gamma$

    change to

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

Predictive Parsers do no Backtracking

- Predict production of non-terminal based on input
  - Limit grammar so 100% prediction accuracy is possible, so that backtracking can be avoided completely!
  - First terminal of every alternative production is unique

    A $\rightarrow$ a B D | b B B
    B $\rightarrow$ c | b c e
    D $\rightarrow$ d

    parsing an input "abced" has no backtracking
  - Left factoring to enable prediction

    A $\rightarrow \alpha\beta \mid \alpha\gamma$

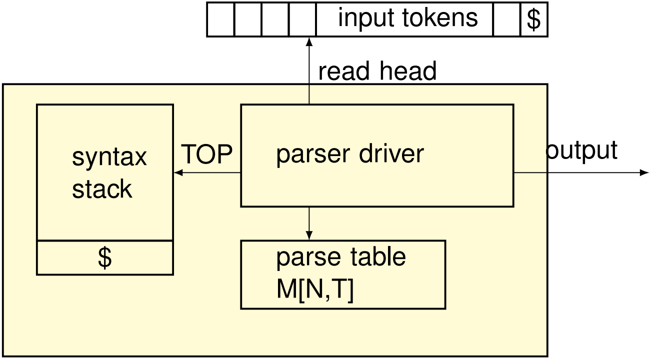
    change to

    A $\rightarrow \alpha$ A'
    A' $\rightarrow \beta \mid \gamma$
- Still does recursive calls, so must eliminate left recursion
  - Recall our hand-coded recursive descent parser

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 — hold right hand side (RHS) of grammar rules
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

	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		$Y \to {*}T$	$Y \to \varepsilon$		$Y \to \varepsilon$	$Y \to \varepsilon$

- Implementation with 2D parse table
  - **First column** lists all non-terminals
  - **First row** lists all possible terminals and $
  - A table entry contains one production
    - One action for each (non-terminal, input) combination
    - No backtracking required

Algorithm for Parsing

X — symbol at the top of the syntax stack

a — current input symbol

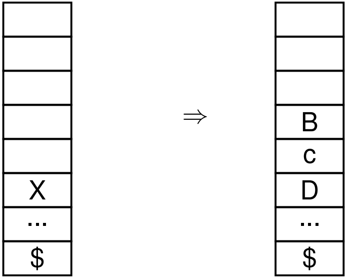
- Parsing based on **(X,a)**
  - If X==a==\$, then
    - parser halts with "success"
  - If X==a!=\$, then
    - pop X from stack **and** advance input head
  - If X!=a, then
    Case (a): if X $\in$ T, then
    - parser halts with "failed", input rejected

    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”



then

Applying LL(1) Parsing to a Grammar

- Given our old grammar

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

  - No left recursion
  - But require left factoring

- After rewriting grammar, we have

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

Using the Parse Table

- To recognize “int * int”

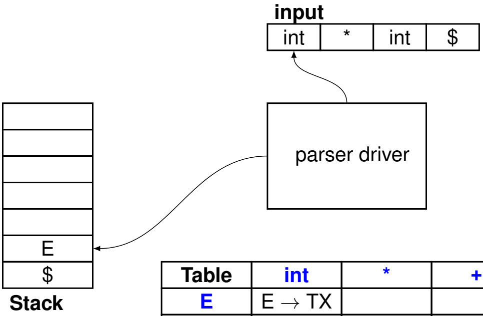


Table	int	*	+	(	)	$
E	$E \rightarrow TX$			$E \rightarrow TX$		
X			$X \rightarrow +E$		$X \rightarrow \varepsilon$	$X \rightarrow \varepsilon$
T	$T \rightarrow$ int Y			$T \rightarrow (E)$		
Y		$Y \rightarrow *T$	$Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

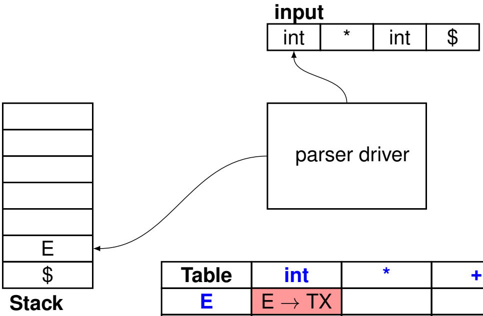


Table	int	*	+	(	)	$
E	$E \rightarrow TX$			$E \rightarrow TX$		
X			$X \rightarrow +E$		$X \rightarrow \varepsilon$	$X \rightarrow \varepsilon$
T	$T \rightarrow$ int Y			$T \rightarrow (E)$		
Y		$Y \rightarrow *T$	$Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

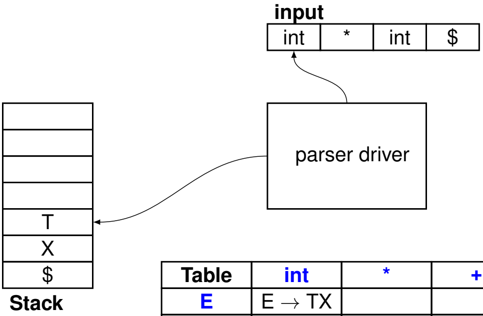


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

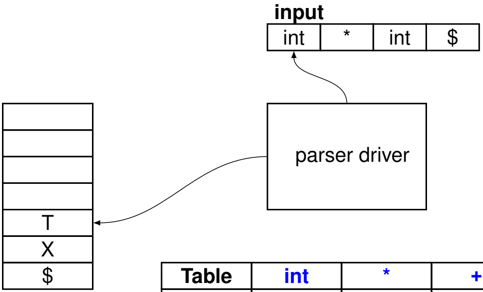


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

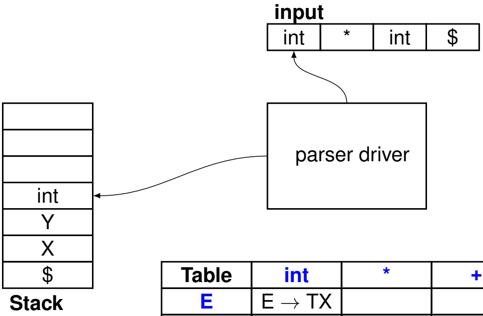


Table	int	*	+	(	)	$
E	$E \to TX$			$E \to TX$		
X			$X \to +E$		$X \to \varepsilon$	$X \to \varepsilon$
T	$T \to$ int Y			$T \to (E)$		
Y		$Y \to *T$	$Y \to \varepsilon$		$Y \to \varepsilon$	$Y \to \varepsilon$

Using the Parse Table

- To recognize “int * int”

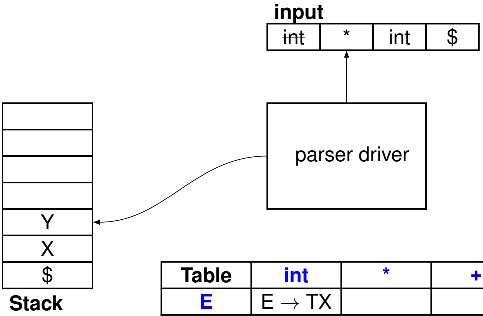


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

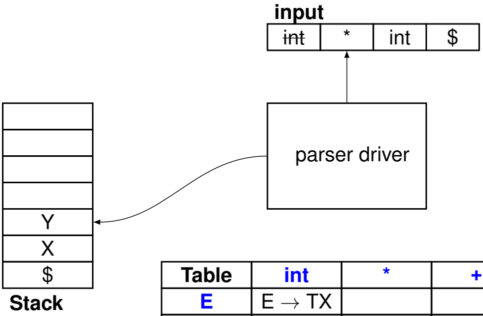


Table	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → *T	Y → ε		Y → ε	Y → ε

Using the Parse Table

- To recognize “int * int”

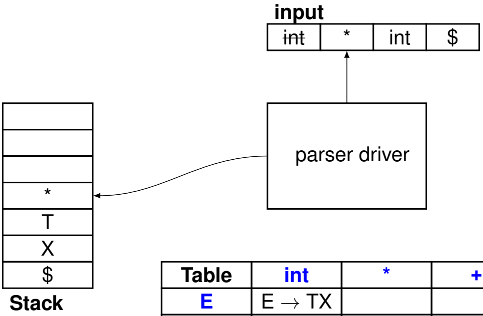


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

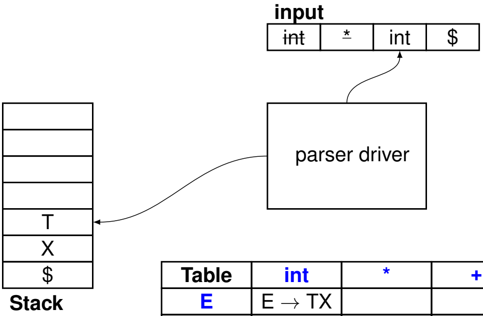


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize "int * int"

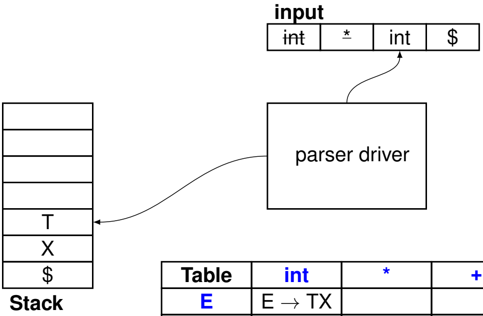


Table	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → *T	Y → ε		Y → ε	Y → ε

Using the Parse Table

- To recognize “int * int”

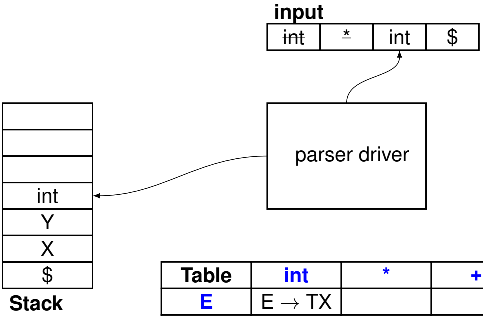


Table	int	*	+	(	)	$
E	$E \rightarrow TX$			$E \rightarrow TX$		
X			$X \rightarrow +E$		$X \rightarrow \varepsilon$	$X \rightarrow \varepsilon$
T	$T \rightarrow \text{int } Y$			$T \rightarrow (E)$		
Y		$Y \rightarrow *T$	$Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

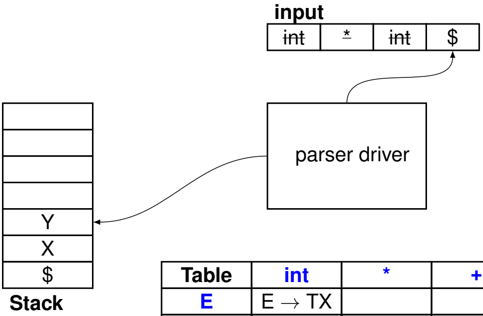


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

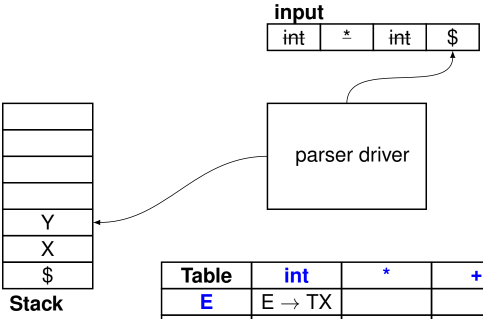


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow\varepsilon$	X $\rightarrow\varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow\varepsilon$		Y $\rightarrow\varepsilon$	Y $\rightarrow\varepsilon$

Using the Parse Table

- To recognize “int * int”

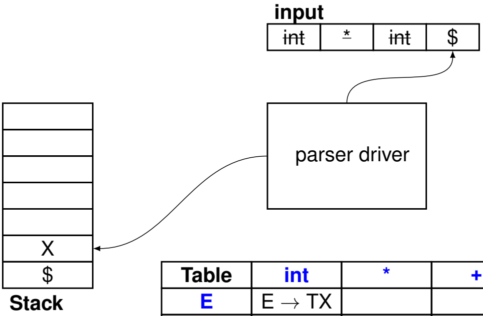


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow$ ε	X $\rightarrow$ ε
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow$ ε		Y $\rightarrow$ ε	Y $\rightarrow$ ε

Using the Parse Table

- To recognize "int * int"

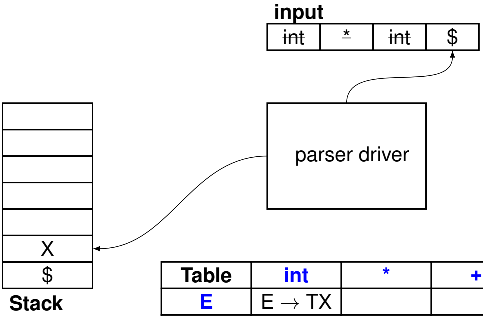


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

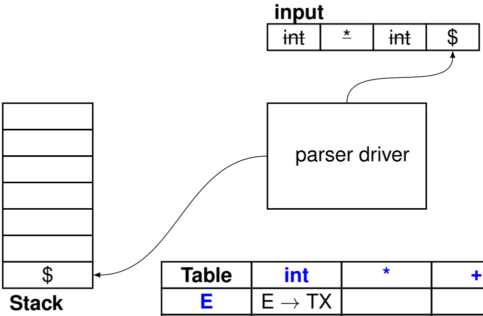
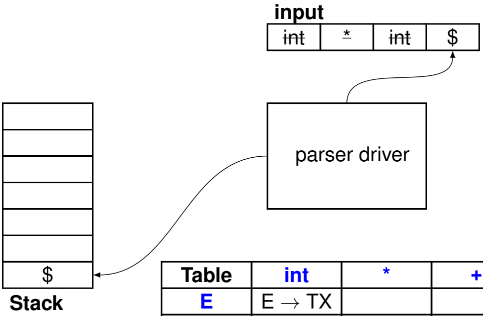


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize "int * int"



Accept!!!

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

Recognition Sequence

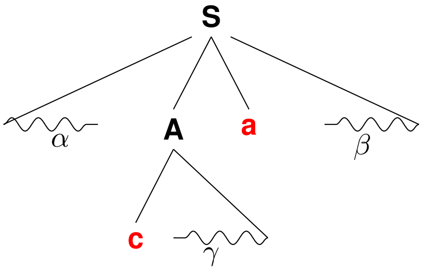
- It is possible to write in a action list

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

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**



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

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

- Why is the Follow Set important?

First(α)

- First(α) = set of terminals that start string of terminals derived from α.

- Apply following rules until no terminal or ε can be added
  1). If t $\in$ T, then First(t)={t}.
  For example First(+)={+}.
  2). If X $\in$ N and X$\rightarrow$ ε exists, then add ε to First(X).
  For example, First(Y) = {*, ε}.
  3). If X $\in$ N and X$\rightarrow$ $Y_1 Y_2 Y_3 \ldots Y_m$, where Y_1, Y_2, Y_3, ... Y_m are non-terminals, then
    - Add (First(Y_1) - ε) to First(X).
    - 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(X).
    - If First(Y_1), ..., First(Y_m) all contain ε, then add ε to First(X).

Follow(α)

- 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 followsing 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 A $\rightarrow$ $\alpha B\beta$, then First(β)-{ε} $\subseteq$ Follow(B)

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

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

Informal Interpretation of First and Follow Sets

- **First** set of X
  - Terminal symbols
  - X $\rightarrow$ Y Z, then First(Y)
  - X $\rightarrow$ ε
- **Follow** set of X
  - \$
  - ... $\rightarrow$ X Y, focus on X
  - Y $\rightarrow$ X, focus on X

For the example

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

- For the first set

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

- For the follow set

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

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
Y	*, ε
X	+, ε
T	(, int
E	(, int

Symbol	Follow
E	$,)
X	$,)
T	$,),+
Y	$,),+

Construction of LL(1) Parse Table

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

Example

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

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

Symbol	Follow
E	$,)
X	$,)
T	$,),+
Y	$,),+

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

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

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

  (2) If (1) fails, the possibilties are...
    - Grammar just needs a little more lookahead
      (May need LL(k) parser where k > 1 or backtracking)
    - Grammar is inherently ambiguous
      (May result in multiple legal derivations)

Ambiguous Grammars

- 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"

  "else" $\in$ First(X)
  First(X)-$\varepsilon \subseteq$ Follow(S)
  X $\rightarrow$ else ... $|$ ε
  "else" $\in$ Follow(X)

- Such grammars are potentially ambiguous

Removing Ambiguity

- To remove ambiguity, it is possible to rewrite the grammar
- For the “if-then-else” example, how would you rewrite it?

  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 arbirary 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 to resolve ambiguity
  - E.g. Always choose X $\rightarrow$ else S over X $\rightarrow$ ε on ’else’ token

LL(1) Summary

- 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?

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 input **t**, use a production rule **A** $\rightarrow \alpha$ where **t** $\in$ **First**(α)
  - For non terminal **A** and input **t**, if **A** $\rightarrow \alpha$ and t $\in$ Follow (A), use the production **A** $\rightarrow \alpha$ where $\varepsilon \in$ **First**(α)

Questions

- What is LL(0)?

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