Syntax Analysis

Comparison with Lexical Analysis

- The second phase of compilation

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

What Parse Tree ?

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

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

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

A Parse Tree Example

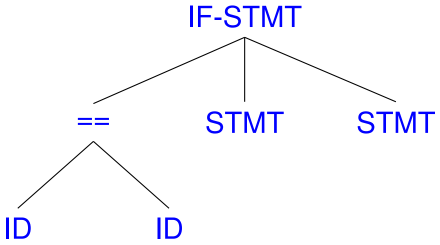
- Code to be compiled:

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

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

    ... IF ID==ID THEN ... ELSE ... FI

  - Desired output:



What Formalism to Use?

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

What Formalism to Use?

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

- RE/FA is not powerful enough

  Example: matching parenthesis: # of "(" equals # of ")"
  - ✔ (x+y)*z
  - ✔ ((x+y)+y)*z

    ...
  - ✔ (...(((x+y)+y)+y)...)

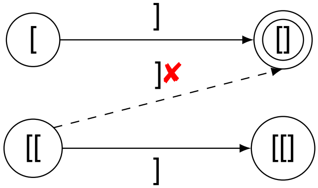
  - ✘ ((x+y)+y)+y)*z

RE/FA is Not Powerful Enough

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

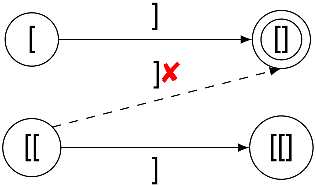
RE/FA is Not Powerful Enough

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



RE/FA is Not Powerful Enough

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



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

Formalism for Syntax Analysis

- We need a more powerful formalism for describing language constructs
  - CFL (context free language) concept
- Before discussing CFL, let us generalize language definition
  - Covers both RE and CFL
  - and more ...

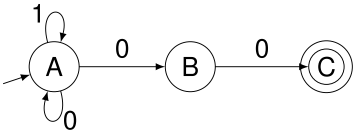
From Grammar to Language

- Recall language definition
  - Language — set of strings over alphabet
    Alphabet: finite set of symbols
    Null string: ε
    Sentences: strings in the language

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

An Example

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



- Grammar G = { T, N, s, δ }

  where T = { 0, 1 }, N = { A, B }, s = A, and
  grammar rule set 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

- A **grammar** consists of 4 components **(T, N, s, δ)**
  - T — set of **terminal** symbols
    - Essentially tokens — those appear in the input string
  - N — set of **non-terminal** symbols
    - Categories of strings impose hierarchical language structure
    - Useful for analysis
      example: declaration, statement, loop, ...
  - s — a special non-terminal **start symbol** that denotes every sentence is derivable 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"
  - to replace LHS with RHS
  - it specifies how to transform one string to another

- $\beta \Rightarrow \alpha$: 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

- A classification of languages based on the form of grammar rules
  - Classify not based on how complex the language is
  - Classify based on how complex the grammar (the describe the language) is

- Four(4) types of grammars:

  Type 0 — recursive grammar
  Type 1 — context sensitive grammar
  Type 2 — context free grammar
  Type 3 — regular grammar

Type 0: Unrestricted/Recursive Grammar

- Type 0 grammar — unrestricted or recursive grammar
  - Form of rules

    $\alpha \rightarrow \beta$

    where $\alpha \in (N \cup T)^+$, $\beta \in (N \cup T)^*$
  - No restrictions on form of grammar rules
  - Example:

    aAB $\rightarrow$ aCD
    aAB $\rightarrow$ aB
    A $\rightarrow$ ε ; empty rule is allowed

Type 1: Context Sensitive Grammar

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

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

    where $A \in N^+$, $\alpha, \beta \in (N \cup T)^*$, $\gamma \in (N \cup T)^+$,
    $|A| \leq |\gamma|$
  - 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
  - No erase rule
    - If there are rules deriving empty string, rewrite to remove empty rule
    - Sometimes loose this restriction to simplify representation

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
  - No erase rule
    - If there are rules deriving empty string, rewrite to remove empty rule
    - Sometimes loose this restriction to simplify representation
- 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 | 0

Differentiate Type 2 and 3 Grammars

- Language L1 = $\{ [^i]^j \;\; | \;\; i, j >= 1\}$
  - Regular grammar

    S → [S | [T
    T →] T |]

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

    S → [S] | [ ]

Differentiate Type 1 and 2 Grammars

- Type 2 grammar (context free)

  S $\rightarrow$ D U
  D $\rightarrow$ int x; | int y;
  U $\rightarrow$ x=1; | y=1;

- Type 1 grammar (context sensitive)

  S $\rightarrow$ D U
  D $\rightarrow$ int x; | int y;
  int x; U $\rightarrow$ int x; x=1;
  int y; U $\rightarrow$ int y; y=1;

What Does a Programming Language Want?

- Language from type 2 grammar
  - S ⇒ DU ⇒ int x; U ⇒ int x; x=1;
  - S ⇒ DU ⇒ int x; U ⇒ int x; y=1;
  - S ⇒ DU ⇒ int y; U ⇒ int y; x=1;
  - S ⇒ DU ⇒ int y; U ⇒ int y; y=1;

- Language from type 1 grammar
  - S ⇒ DU ⇒ int x; U ⇒ int x; x=1;
  - S ⇒ DU ⇒ int y; U ⇒ int y; y=1;

What Does a Programming Language Want?

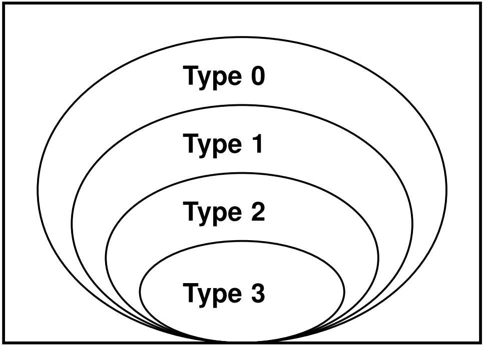
- Language from type 2 grammar
  - S $\Rightarrow$ DU $\Rightarrow$ int x; U $\Rightarrow$ int x; x=1;
  - S $\Rightarrow$ DU $\Rightarrow$ int x; U $\Rightarrow$ int x; y=1;
  - S $\Rightarrow$ DU $\Rightarrow$ int y; U $\Rightarrow$ int y; x=1;
  - S $\Rightarrow$ DU $\Rightarrow$ int y; U $\Rightarrow$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

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

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

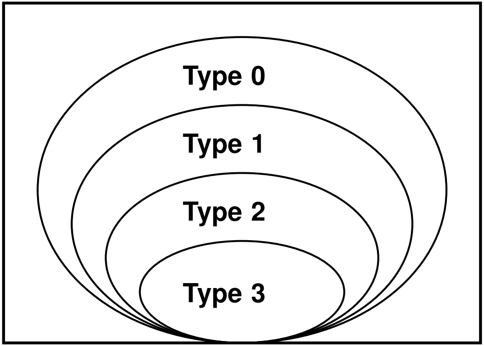
Language Classification

- Regular Grammar $\subseteq$ CFG $\subseteq$ CSG $\subseteq$ Recursive Grammar



Language Classification

- Regular Grammar $\subseteq$ CFG $\subseteq$ CSG $\subseteq$ 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

Grammar and Syntax Analysis

- Grammar is used to derive string or construct parser

- A **derivation** is a sequence of applications of rules
  - Starting from the **start symbol**
  - S ⇒ ... ⇒ ... ⇒ ... ⇒ (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
  $E \Rightarrow E * E \Rightarrow E * E + E \Rightarrow E * E + id \Rightarrow E * E + E + id \Rightarrow ...$
  $\Rightarrow id * id + id * id$

Parse Trees

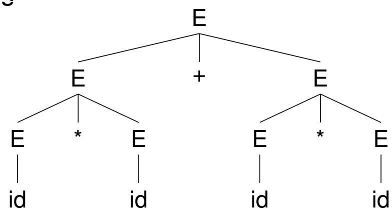
- Parse tree can
  - filter out the order of replacement
  - describe hierarchy
- Parse tree
  - Internal nodes are non-terminals
  - Leaves are terminals

  - Same parse tree for the previous rightmost/leftmost derivations

Parse Trees

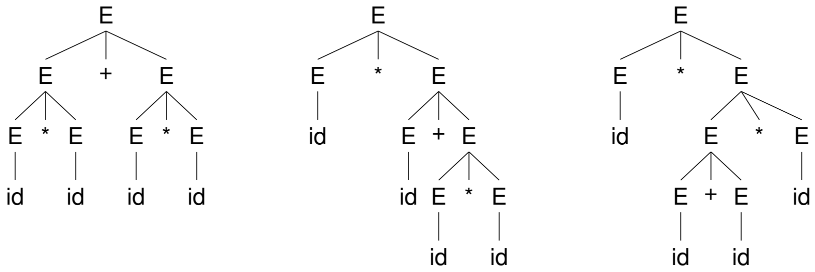
- Parse tree can
  - filter out the order of replacement
  - describe hierarchy
- Parse tree
  - Internal nodes are non-terminals
  - Leaves are terminals

  - Same parse tree for the previous rightmost/leftmost derivations



Different Parse Trees

- Consider the string
  id * id + id * id
  can draw 3 different trees



Ambiguity

- A grammar G is **ambiguous** if
  - there exist a string *str* $\in$ L(G) such that
  - more than one parse trees derive *str*
- In practice, we prefer unambiguous grammars
- Ambiguity is the property of a grammar and not the language
  - It is possible to rewrite the grammar to remove abmiguity

How to Remove Ambiguity?

- Method I: to specify **precedence**
  - build precedence into grammar, have different non-terminal for each precedence level
    - Lowest level — highest in the tree (lowest precedence)
    - Highest level — lowest in the tree
    - Same level — same precedence

- For the previous example,

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

  rewrite it to

  $$E \rightarrow E + T \mid E - T \mid T$$
  $$T \rightarrow T * F \mid T / F \mid F$$
  $$F \rightarrow P \wedge F \mid P$$
  $$P \rightarrow id \mid const \mid (E)$$

How to Remove Ambiguity?

- Method II: to specify **associativity**
  - when recursion is allowed, we need to specify associativity

- For the previous example,

  $E \rightarrow E - E$ .... ; allows both left and right associativity

  rewrite it to

  $E \rightarrow E + T$ ... ; only left associativity

  $F \rightarrow P \wedge F$ ... ; only right associativity

From Grammar to Syntax Analysis

- We discussed grammar from the point of view of derivation

- What is **syntax analysis**?
  - To process an input string for a given grammar, and compose the derivation if the string is in the language
  - Two subtasks
    - to determine if string in the language or not
    - to construct the parse tree

From Grammar to Syntax Analysis

- We discussed grammar from the point of view of derivation

- What is **syntax analysis**?
  - To process an input string for a given grammar, and compose the derivation if the string is in the language
  - Two subtasks
    - to determine if string in the language or not
    - to construct the parse tree

- Is that possible to construct such a parser?

BNF and Parsing

- Backus Naur Form (BNF) is an extension of general CFG
  - ε is allowed — usually for recursion — can be eliminated
  - use $*$ to indicate recursion or structure

    e.g. A $\rightarrow$ A * B C d
  - use $|$ for alternative rules
  - use upper case letters (e.g. A, B) or <class> for **non-terminals**
  - use lower case leeters (e.g. a, b) for **terminals**
- **BNF Properties**:
  - Any BNF grammar has a decidable parsing program
  - It is **decidable** if a string is in the language or not
  - It is **undecidable** if an arbitrary BNF grammar is ambiguous
  - It is **undecidable** if two grammars generate the same language

Types of Parsers

- Universal parser
  - Can parse any BNF grammar e.g. Early's algorithm
  - Powerful but extremely inefficient

- Top-down parser
  - It is goal-directed, expands the start symbol to the given sentence
  - Only works for certain class of grammars
  - To start from the root of the parse tree and reach leaves
  - Find leftmost derivation
  - Can be implemented efficiently by hand

Types of Parsers (cont.)

- Bottom-up parser
  - It tries to *reduce* input string to the start symbol
  - Works for wider class of grammars
  - Starts at leaves and build tree in bottom-up fashion
  - Find reverse order of the rightmost derivation
  - Automated tool generates it automatically

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 ignore some details
  - internal nodes may contain terminal symbols

An Example

- Consider the grammar

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

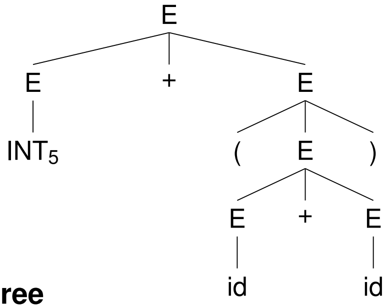
  and an input

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

- After lexical analysis, we have a sequence of tokens

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

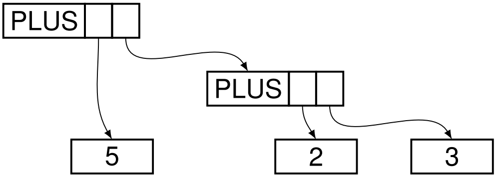
Parse Tree of the Input



- 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

- We prefer an **Abstract Syntax Tree (AST)** as follows



- **AST** also captures the nested structure
- **AST** abstracts from the concrete syntax
- **AST** is more compact and easier to use

- **AST** is an important data structure in a compiler

How to Construct AST?

- Introduce the concept of **semantic actions**

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

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

$$\mathbf{X} \quad \rightarrow \quad \mathbf{Y_1 Y_2 ... Y_n} \qquad \mathbf{\{ \text{ actions } \}}$$

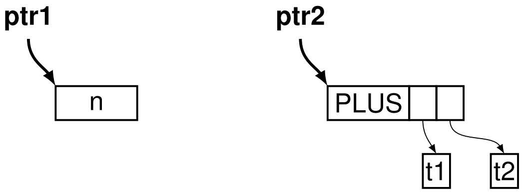
actions may define or use X.ast, Y_i.ast ($1 \leq i \leq n$)

Example

- For the previous example, we have

$$
\begin{array}{llll}
E & \rightarrow & \text{int} & \{\ E.ast = mkleaf(int.lval)\ \} \\
 & | & E1 + E2 & \{\ E.ast = mkplus(E1.ast, E2.ast)\ \} \\
 & | & (E1) & \{\ E.ast = E1.ast\ \}
\end{array}
$$

- 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

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

we build in a bottom-up fashion

AST Construction Steps

with input INT$_5$ ‘+’ ‘(’ INT$_2$ ‘+’ INT$_3$ ‘)’

we build in a bottom-up fashion

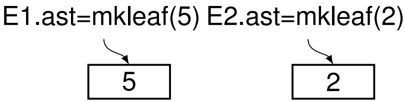
E1.ast=mkleaf(5)

$\boxed{5}$

AST Construction Steps

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

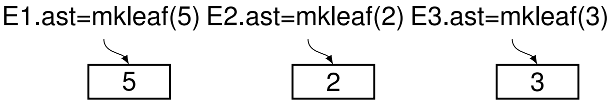
we build in a bottom-up fashion



AST Construction Steps

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

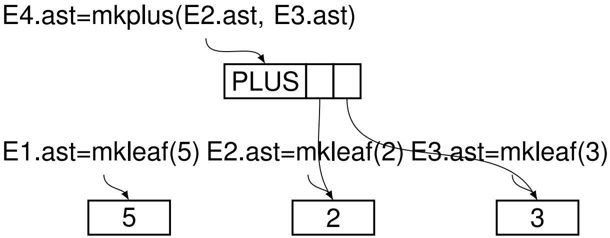
we build in a bottom-up fashion



AST Construction Steps

with input INT_5 ‘+’ ‘(’ INT_2 ‘+’ INT_3 ‘)’

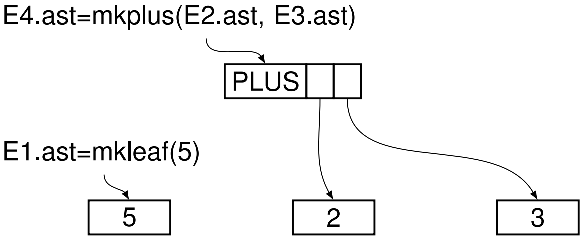
we build in a bottom-up fashion



AST Construction Steps

with input INT$_5$ '+' '(' INT$_2$ '+' INT$_3$ ')'

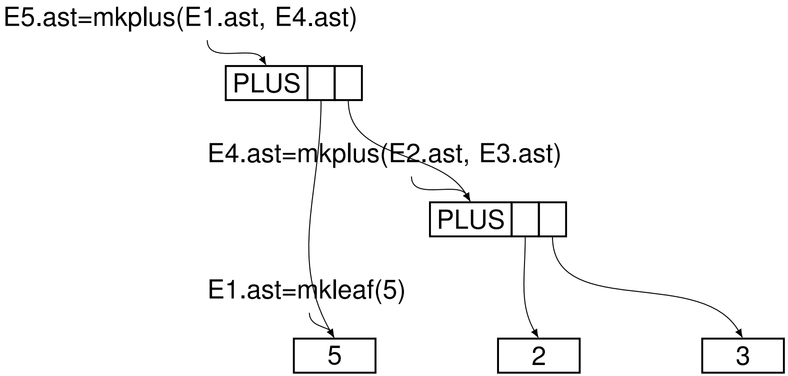
we build in a bottom-up fashion



AST Construction Steps

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

we build in a bottom-up fashion

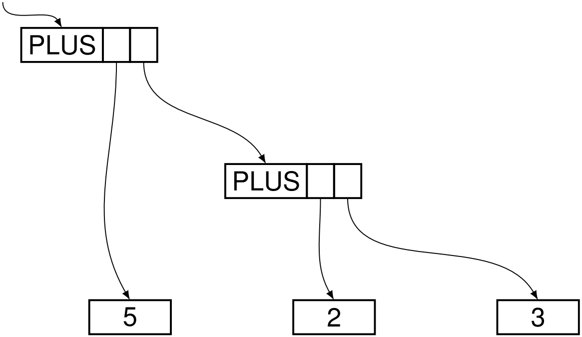


AST Construction Steps

with input INT$_5$ ‘+’ ‘(’ INT$_2$ ‘+’ INT$_3$ ‘)’

we build in a bottom-up fashion

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

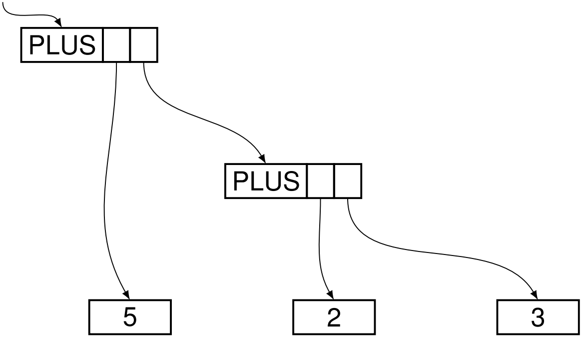


AST Construction Steps

with input INT$_5$ ‘+’ ‘(’ INT$_2$ ‘+’ INT$_3$ ‘)’

we build in a bottom-up fashion

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



Summary

- We specify the syntax structure using CFG
  - the programming language itself is not context free
- A parser can
  - ... 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

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

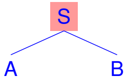
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { acbd }

Leftmost Derivation:

$S \Rightarrow AB$ (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

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

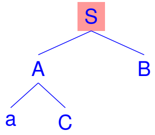
$D \rightarrow d \qquad C \rightarrow c$

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

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

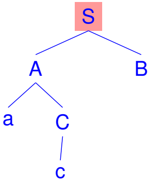
$D \rightarrow d \qquad C \rightarrow c$

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

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

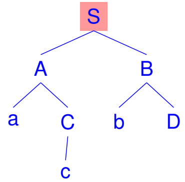
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { acbd }

Leftmost Derivation:

$S \Rightarrow AB$ (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

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

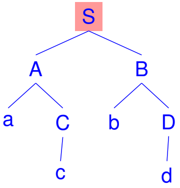
$D \rightarrow d \qquad C \rightarrow c$

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

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

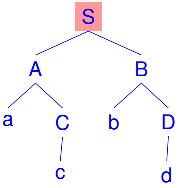
$D \rightarrow d \qquad C \rightarrow c$

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)

a c b d

Example

Consider a CFG grammar G

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

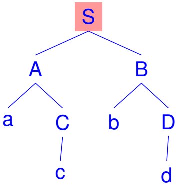
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { acbd }

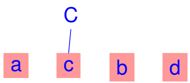
Leftmost Derivation:

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

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

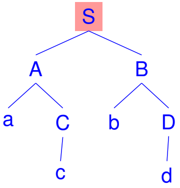
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { acbd }

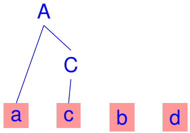
Leftmost Derivation:

$S \Rightarrow$ 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

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

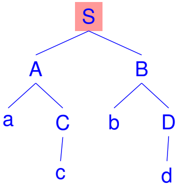
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { acbd }

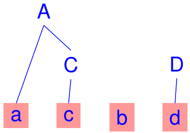
Leftmost Derivation:

$S \Rightarrow AB$ (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

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

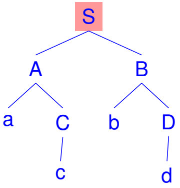
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { acbd }

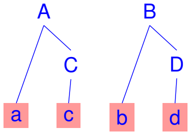
Leftmost Derivation:

$S \Rightarrow AB$ (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

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

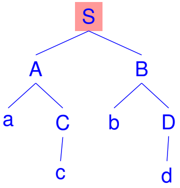
$D \rightarrow d \qquad C \rightarrow c$

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

L(G) = { acbd }

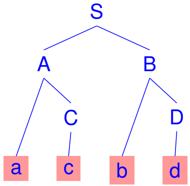
Leftmost Derivation:

$S \Rightarrow AB$ (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
  - Simple to implement, use backtracking
- Predictive parser
  - Predict the rule based on the 1st m symbols without backtracking
  - Restrictions on the grammar to avoid backtracking
- LL(k) — predicative parser for LL(k) grammar
  - Non recursive and only k symbol look ahead
  - Table driven — efficient

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

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 $E \rightarrow T$
– 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 ⇒ T ⇒ (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)

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

- 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 up 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 up 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 up T→int
– “int” matches input “int”
– however, we expect more tokens
– failure, backtrack one level

Parsing Sequence (using Backtracking)

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

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

$\Rightarrow \text{int} * T$

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

Parsing Sequence (using Backtracking)

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

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

$\Rightarrow \text{int} * T \Rightarrow \text{int} * (E)$

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

$\cancel{\Rightarrow int}$

- pick up T→int
- "int" matches input "int"
- however, we expect more tokens
- failure, backtrack one level

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

- pick up T→int * T
- pick up T→int * (E)
- "(" matches input "int"
- failure, backtrack one level

Parsing Sequence (using Backtracking)

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

Parsing Sequence (using Backtracking)

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

Implementation

- Create a procedure for each non-terminal
  1. Checks if input symbol matches a terminal symbol in the grammar rule
  2. Calls other procedure when non-terminals are part of the rule
  3. If end of procedure is reached, success is reported to the caller

Sample Code

- (Rewrite the rule a bit, will discuss the reason)

$$E \rightarrow T \{ + E \}$$
$$T \rightarrow int \{ * T \} \;|\; (E)$$

```
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();
       fetchNext();
       if (sym!=RightParenNum)
         perror("error");
       fetchNext();
    }
 }
```

Left Recursion Problem

- The previous discussion does not work if grammar is left recursive
  - Right recursive is okay
- Why left recursion is a problem?

  $A \rightarrow A\ b \mid c$

  We may have

  $A \Rightarrow A\ b \Rightarrow A\ b\ b \ldots$

  the sentential form keeps growing without consuming any input symbol

Left Recursion Problem

- The previous discussion does not work if grammar is left recursive
  - Right recursive is okay
- Why left recursion is a problem?

  $A \rightarrow A\ b \mid c$

  We may have

  $A \Rightarrow A\ b \Rightarrow A\ b\ b \ldots$

  the sentential form keeps growing without consuming any input symbol
- Rewrite the grammar to represent the same language
  - What language does this grammar generate?

Remove Left Recursion

- In general, we can eliminate all immediate left recursion

  $$A \rightarrow A\ x \mid y$$

  change to

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

- Not all left recursion is immediate

  may be hidden in multiple production rules

  $$A \rightarrow BC \mid D$$
  $$B \rightarrow AE \mid F$$

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

  ... (not required for this course)

Summary of Recursive Descent

- Recursive descent is a simple and general parsing strategy
  - Left-recursion must be eliminated first
    - Can be eliminated automatically
  - It is not popular because of its inefficiency
    - Backtracking re-parses the string
    - Undo semantic actions may be difficult !!!
- Techniques used in practice do no backtracking

  ... at the cost of restricting the class of grammar

Predicative Parsers

- To avoid backtracking: for a given input symbol and given non-terminal, choose the alternative **appropriately**
  - The first terminal of every alternative in a 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 predication

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

    change to

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

Predicative Parsers

- To avoid backtracking: for a given input symbol and given non-terminal, choose the alternative **appropriately**
  - The first terminal of every alternative in a 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 predication

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

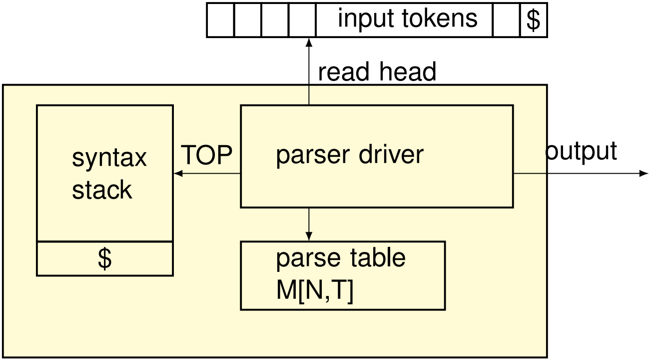
    change to

    A $\rightarrow \alpha$ A'
    A' $\rightarrow \beta \mid \gamma$
- For predicative parsers, must eliminate left recursion
  - Recall our sample C code

LL(k) Parsers

- LL(k)
  - L — left to right scan
  - L — leftmost derivation
  - k — k symbols of lookahead
  - in practice, k = 1
- It is table-driven and efficient

Parser Structure



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)**

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		$Y \rightarrow *T$	$Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$

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

Algorithm for Parsing

X — symbol at the top of the syntax stack

a — current input symbol

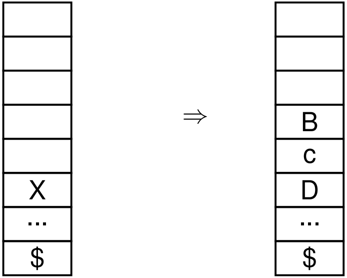
- Parsing based on **(X,a)**
  - If X==a==\$, then
    - parser halts with "success"
  - If X==a!=\$, then
    - pop X from stack **and** advance input head
  - If X!=a, then
    Case (a): if X $\in$ T, then
    - parser halts with "failed", input rejected

    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”



Applicable Grammars

- As we discussed, remove left recursive and perform left factoring
- Given the grammar

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

  - No left recursion
  - But require left factoring
- After rewriting grammar, we have

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

Using the Parse Table

- To recognize “int * int”

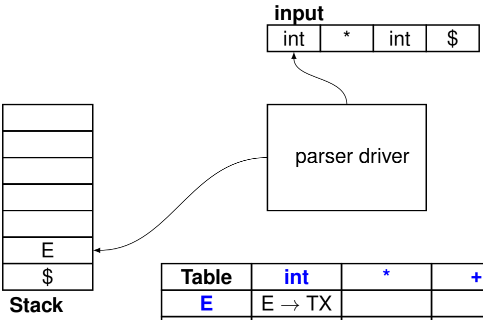


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

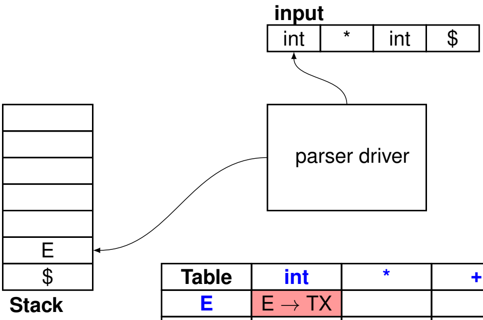


Table	int	*	+	(	)	$
E	$E \to TX$			$E \to TX$		
X			$X \to +E$		$X \to \varepsilon$	$X \to \varepsilon$
T	$T \to$ int Y			$T \to (E)$		
Y		$Y \to {*}T$	$Y \to \varepsilon$		$Y \to \varepsilon$	$Y \to \varepsilon$

Using the Parse Table

- To recognize “int * int”

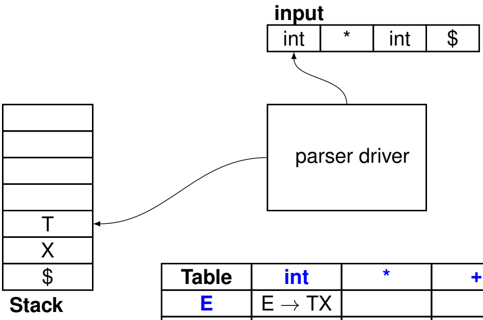


Table	int	*	+	(	)	$
E	$E \rightarrow TX$			$E \rightarrow TX$		
X			$X \rightarrow +E$		$X \rightarrow \varepsilon$	$X \rightarrow \varepsilon$
T	$T \rightarrow \text{int } Y$			$T \rightarrow (E)$		
Y		$Y \rightarrow *T$	$Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

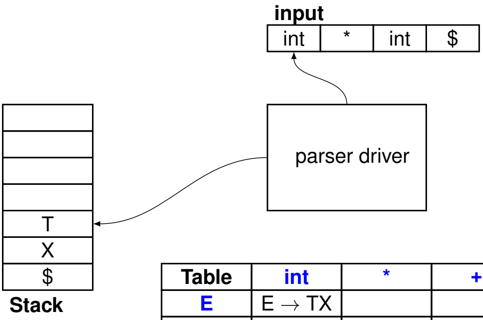


Table	int	*	+	(	)	$
E	$E \to TX$			$E \to TX$		
X			$X \to +E$		$X \to \varepsilon$	$X \to \varepsilon$
T	$T \to$ int Y			$T \to (E)$		
Y		$Y \to *T$	$Y \to \varepsilon$		$Y \to \varepsilon$	$Y \to \varepsilon$

Using the Parse Table

- To recognize "int * int"

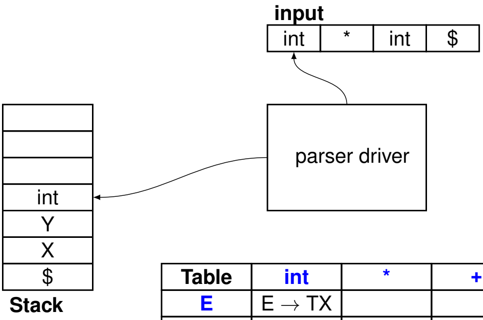


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow \varepsilon$	X $\rightarrow \varepsilon$
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow \varepsilon$		Y $\rightarrow \varepsilon$	Y $\rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

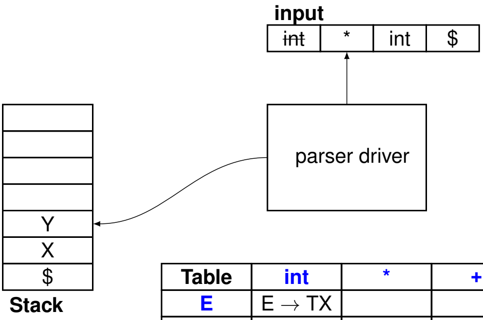


Table	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → *T	Y → ε		Y → ε	Y → ε

Using the Parse Table

- To recognize “int * int”

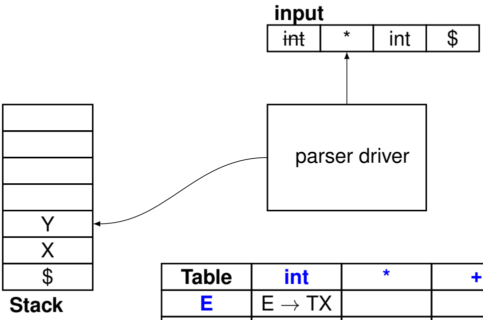


Table	int	*	+	(	)	$
E	$E \rightarrow TX$			$E \rightarrow TX$		
X			$X \rightarrow +E$		$X \rightarrow \varepsilon$	$X \rightarrow \varepsilon$
T	$T \rightarrow \text{int } Y$			$T \rightarrow (E)$		
Y		$Y \rightarrow *T$	$Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

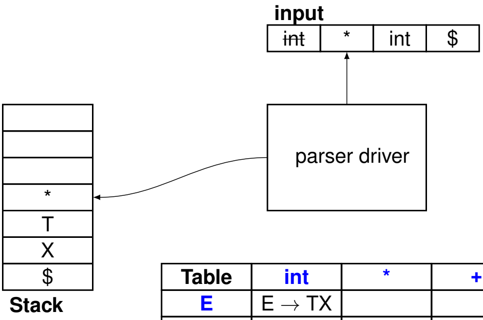


Table	int	*	+	(	)	$
E	$E \rightarrow TX$			$E \rightarrow TX$		
X			$X \rightarrow +E$		$X \rightarrow \varepsilon$	$X \rightarrow \varepsilon$
T	$T \rightarrow \text{int } Y$			$T \rightarrow (E)$		
Y		$Y \rightarrow *T$	$Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$

Using the Parse Table

- To recognize "int * int"

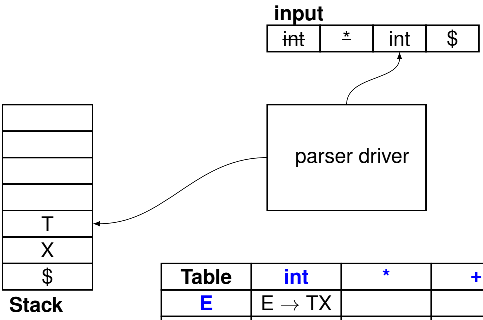


Table	int	*	+	(	)	$
E	E → TX			E → TX		
X			X → +E		X → ε	X → ε
T	T → int Y			T → (E)		
Y		Y → *T	Y → ε		Y → ε	Y → ε

Using the Parse Table

- To recognize “int * int”

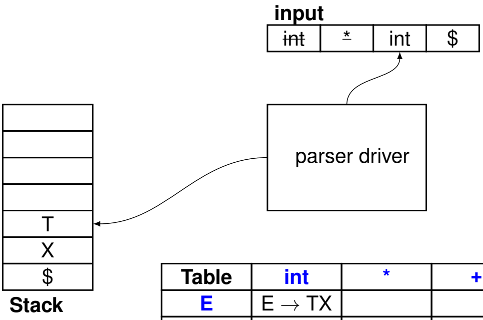


Table	int	*	+	(	)	$
E	$E \rightarrow TX$			$E \rightarrow TX$		
X			$X \rightarrow +E$		$X \rightarrow \varepsilon$	$X \rightarrow \varepsilon$
T	$T \rightarrow int\ Y$			$T \rightarrow (E)$		
Y		$Y \rightarrow *T$	$Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

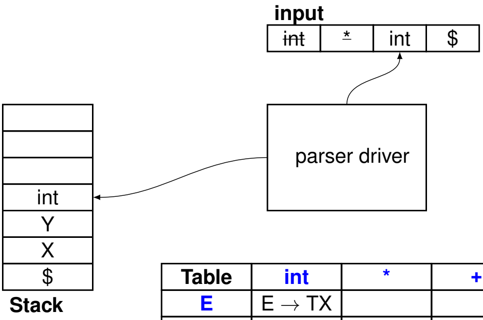


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow$ ε	X $\rightarrow$ ε
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow$ ε		Y $\rightarrow$ ε	Y $\rightarrow$ ε

Using the Parse Table

- To recognize “int * int”

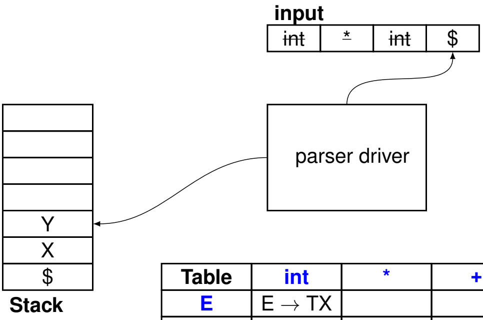


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow$ ε	X $\rightarrow$ ε
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow$ ε		Y $\rightarrow$ ε	Y $\rightarrow$ ε

Using the Parse Table

- To recognize “int * int”

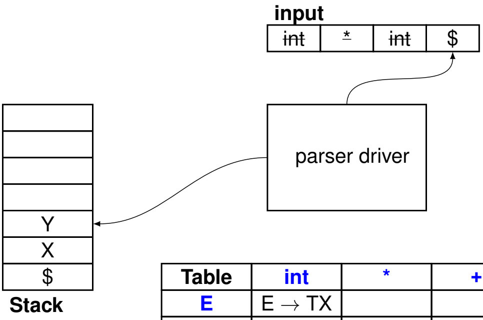


Table	int	*	+	(	)	$
E	$E \rightarrow TX$			$E \rightarrow TX$		
X			$X \rightarrow +E$		$X \rightarrow \varepsilon$	$X \rightarrow \varepsilon$
T	$T \rightarrow int\ Y$			$T \rightarrow (E)$		
Y		$Y \rightarrow *T$	$Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”

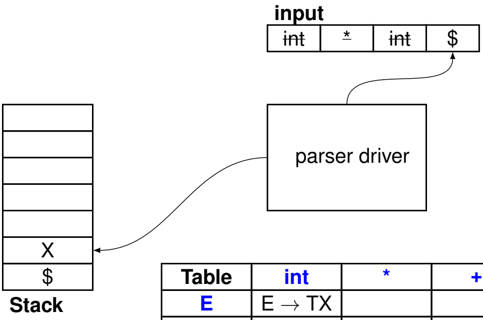


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow$ ε	X $\rightarrow$ ε
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow$ ε		Y $\rightarrow$ ε	Y $\rightarrow$ ε

Using the Parse Table

- To recognize "int * int"

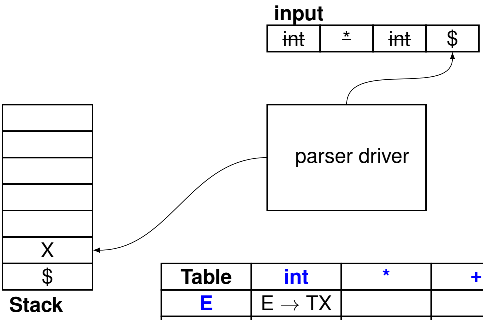


Table	int	*	+	(	)	$
E	E $\rightarrow$ TX			E $\rightarrow$ TX		
X			X $\rightarrow$ +E		X $\rightarrow$ ε	X $\rightarrow$ ε
T	T $\rightarrow$ int Y			T $\rightarrow$ (E)		
Y		Y $\rightarrow$ *T	Y $\rightarrow$ ε		Y $\rightarrow$ ε	Y $\rightarrow$ ε

Using the Parse Table

- To recognize “int * int”

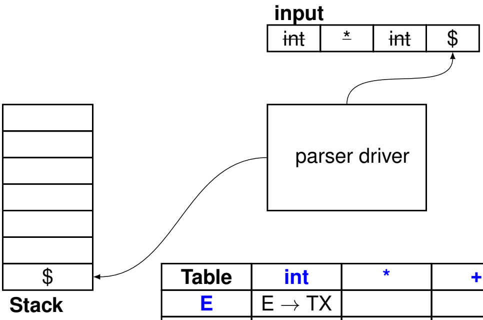
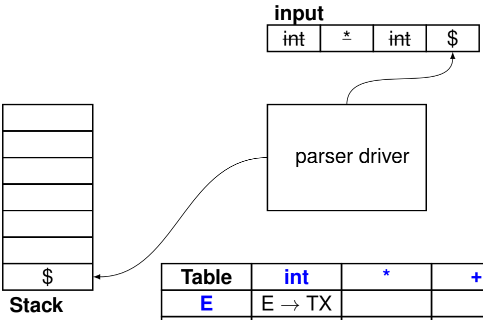


Table	int	*	+	(	)	$
E	$E \rightarrow TX$			$E \rightarrow TX$		
X			$X \rightarrow +E$		$X \rightarrow \varepsilon$	$X \rightarrow \varepsilon$
T	$T \rightarrow$ int Y			$T \rightarrow (E)$		
Y		$Y \rightarrow *T$	$Y \rightarrow \varepsilon$		$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$

Using the Parse Table

- To recognize “int * int”



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

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

First(α)

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

- Apply followsing 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 $(\sum_{1 \le i \le k} First(Y_i) - \varepsilon)$ to First(X).
     - If First(Y_1), ..., First(Y_m) all contain ε, then add ε to First(X).

Follow(α)

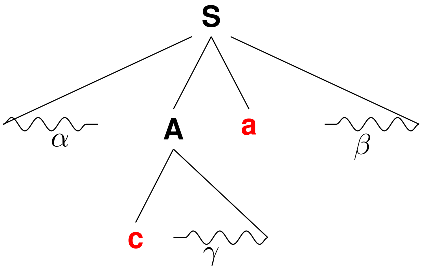
- Follow(α) = $\{t|S \Rightarrow *\alpha t\beta\}$

  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)

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



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

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

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

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	$,),+

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

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 gets 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

Ambiguous Grammars

- Some grammars may need more than one lookahead (k)
- However, some grammars are not LL regardless of how the grammar is changed

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

Removing Ambiguity

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

Removing Ambiguity

- To remove ambiguity, it is possible to rewrite the grammar to remove ambiguity
- For the “if-then-else” example, how to rewrite ?
- However, by changing the grammar,
  - it might make the other phases of the compiler more difficult
  - it becomes harder to determine semantics and generate code
  - it is less appealing to programmers

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?
  - Space stack is smaller than the input (in case 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 ?