

Parser: Syntactic Analysis

Readings: EAC2 Chapter 3

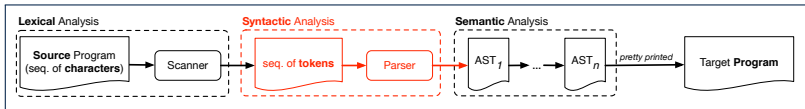


EECS4302 M:
Compilers and Interpreters
Winter 2020

CHEN-WEI WANG

Parser in Context

- Recall:



- Treats the input programs as a **a sequence of classified tokens/words**
- Applies rules **parsing** token sequences as **abstract syntax trees (ASTs)** [**syntactic** analysis]
- Upon termination:
 - Reports token sequences not derivable as ASTs
 - Produces an **AST**
- No longer considers **every character** in input program.
- Derivable** token sequences constitute a **context-free language (CFL)**.

Context-Free Languages: Introduction

- We have seen **regular languages** :
 - Can be described using *finite automata* or *regular expressions*.
 - Satisfy the *pumping lemma*.
- Languages with a *recursive* structure are provably *non-regular*.
e.g., $\{0^n 1^n \mid n \geq 0\}$
- **Context-free grammars (CFG's)** are used to describe strings that can be generated in a *recursive* fashion.
- **Context-free languages (CFL's)** are:
 - Languages that can be described using CFG's.
 - A proper superset of the set of regular languages.

CFG: Example (1.1)

- The language that we previously proved as *non-regular*

$$\{0^n \# 1^n \mid n \geq 0\}$$

can be described using the following **grammar** :

$$A \rightarrow 0A1$$

$$A \rightarrow B$$

$$B \rightarrow \#$$

- A grammar contains a collection of *substitution* or *production* rules, where:
 - A *terminal* is a word $w \in \Sigma^*$ (e.g., 0, 1, etc.).
 - A *variable* or *non-terminal* is a word $w \notin \Sigma^*$ (e.g., A, B, etc.).
 - A *start variable* occurs on the LHS of the topmost rule (e.g., A).

CFG: Example (1.2)

- Given a grammar, generate a string by:
 1. Write down the *start variable*.
 2. Choose a production rule where the *start variable* appears on the LHS of the arrow, and *substitute* it by the RHS.
 3. There are two cases of the re-written string:
 - 3.1 It contains *no* variables, then you are done.
 - 3.2 It contains *some* variables, then *substitute* each variable using the relevant *production rules*.
 4. Repeat Step 3.
- e.g., We can generate an *infinite* number of strings from

$$A \rightarrow 0A1$$

$$A \rightarrow B$$

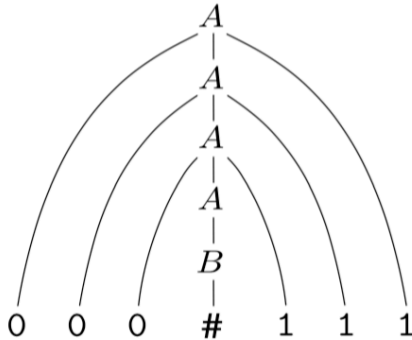
$$B \rightarrow \#$$

- $A \Rightarrow B \Rightarrow \#$
- $A \Rightarrow 0A1 \Rightarrow 0B1 \Rightarrow 0\#1$
- $A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 00B11 \Rightarrow 00\#11$
- ...

CFG: Example (1.2)

Given a CFG, the *derivation* of a string can be shown as a *parse tree*.

e.g., The derivation of 000#111 has the parse tree



CFG: Example (2)

Design a CFG for the following language:

$$\{w \mid w \in \{0, 1\}^* \wedge w \text{ is a palidrome}\}$$

e.g., 00, 11, 0110, 1001, *etc.*

$$P \rightarrow \epsilon$$

$$P \rightarrow 0$$

$$P \rightarrow 1$$

$$P \rightarrow 0P0$$

$$P \rightarrow 1P1$$

CFG: Example (3)

Design a CFG for the following language:

$$\{ww^R \mid w \in \{0, 1\}^*\}$$

e.g., 00, 11, 0110, *etc.*

$$P \rightarrow \epsilon$$

$$P \rightarrow 0P0$$

$$P \rightarrow 1P1$$

CFG: Example (4)

Design a CFG for the set of binary strings, where each block of 0's followed by at least as many 1's.

e.g., 000111, 0001111, *etc.*

- We use S to represent one such string, and A to represent each such block in S .

$S \rightarrow \epsilon$ {BC of S }

$S \rightarrow AS$ {RC of S }

$A \rightarrow \epsilon$ {BC of A }

$A \rightarrow 01$ {BC of A }

$A \rightarrow 0A1$ {RC of A : equal 0's and 1's}

$A \rightarrow A1$ {RC of A : more 1's}

CFG: Example (5.1) Version 1

Design the grammar for the following small expression language, which supports:

- Arithmetic operations: $+$, $-$, $*$, $/$
- Relational operations: $>$, $<$, $>=$, $<=$, $==$, $/=$
- Logical operations: `true`, `false`, `!`, `&&`, `||`, `=>`

Start with the variable *Expression*.

- There are two possible versions:
 1. All operations are mixed together. [e.g., $(1 + \text{true})/\text{false}$]
 2. Relevant operations are grouped together.
Try both!

CFG: Example (5.2) Version 1

<i>Expression</i>	→	<i>IntegerConstant</i>
		<i>-IntegerConstant</i>
		<i>BooleanConstant</i>
		<i>BinaryOp</i>
		<i>UnaryOp</i>
		<i>(Expression)</i>
<i>IntegerConstant</i>	→	<i>Digit</i>
		<i>Digit IntegerConstant</i>
<i>Digit</i>	→	0 1 2 3 4 5 6 7 8 9
<i>BooleanConstant</i>	→	TRUE
		FALSE

CFG: Example (5.3) Version 1

BinaryOp → *Expression* + *Expression*
| *Expression* - *Expression*
| *Expression* * *Expression*
| *Expression* / *Expression*
| *Expression* && *Expression*
| *Expression* || *Expression*
| *Expression* => *Expression*
| *Expression* == *Expression*
| *Expression* /= *Expression*
| *Expression* > *Expression*
| *Expression* < *Expression*

UnaryOp → ! *Expression*

CFG: Example (5.4) Version 1

However, Version 1 of CFG:

- **Parses** string that requires further **semantic analysis** (e.g., type checking):
e.g., $3 \Rightarrow 6$
- Is **ambiguous**, meaning that a string may have more than one ways to interpret it.
e.g., Draw the **parse tree(s)** for $3 * 5 + 4$

CFG: Example (5.5) Version 2

Expression → *ArithmeticOp*
| *RelationalOp*
| *LogicalOp*
| (*Expression*)

IntegerConstant → *Digit*
| *Digit IntegerConstant*

Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

BooleanConstant → TRUE
| FALSE

CFG: Example (5.6) Version 2

<i>ArithmeticOp</i>	→	<i>ArithmeticOp</i> + <i>ArithmeticOp</i> <i>ArithmeticOp</i> - <i>ArithmeticOp</i> <i>ArithmeticOp</i> * <i>ArithmeticOp</i> <i>ArithmeticOp</i> / <i>ArithmeticOp</i> (<i>ArithmeticOp</i>) <i>IntegerConstant</i> - <i>IntegerConstant</i>
<i>RelationalOp</i>	→	<i>ArithmeticOp</i> == <i>ArithmeticOp</i> <i>ArithmeticOp</i> /= <i>ArithmeticOp</i> <i>ArithmeticOp</i> > <i>ArithmeticOp</i> <i>ArithmeticOp</i> < <i>ArithmeticOp</i>
<i>LogicalOp</i>	→	<i>LogicalOp</i> && <i>LogicalOp</i> <i>LogicalOp</i> <i>LogicalOp</i> <i>LogicalOp</i> => <i>LogicalOp</i> ! <i>LogicalOp</i> (<i>LogicalOp</i>) <i>RelationalOp</i> <i>BooleanConstant</i>

CFG: Example (5.7) Version 2

However, Version 2 of CFG:

- Eliminates some cases for further semantic analysis:
e.g., $(1 + 2) \Rightarrow (5 / 4)$ [no parse tree]
- Still *Parses* string that might require further *semantic analysis* :
e.g., $(1 + 2) / (5 - (2 + 3))$
- Is *ambiguous* , meaning that a string may have more than one ways to interpret it.
e.g., Draw the *parse tree(s)* for $3 * 5 + 4$

CFG: Formal Definition (1)

- A **context-free grammar (CFG)** is a 4-tuple (V, Σ, R, S) :

- V is a finite set of **variables**.
- Σ is a finite set of **terminals**.
- R is a finite set of **rules** s.t.

$$[V \cap \Sigma = \emptyset]$$

$$R \subseteq \{v \rightarrow s \mid v \in V \wedge s \in (V \cup \Sigma)^*\}$$

- $S \in V$ is the **start variable**.
- Given strings $u, v, w \in (V \cup \Sigma)^*$, variable $A \in V$, and a rule $A \rightarrow w$:

- $uAv \Rightarrow uwv$ means that uAv **yields** uwv .

- $u \xRightarrow{*} v$ means that u **derives** v , if:

- $u = v$; or
- $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_k \Rightarrow v$

[a yield sequence]

- Given a CFG $G = (V, \Sigma, R, S)$, the language of G

$$L(G) = \{w \in \Sigma^* \mid S \xRightarrow{*} w\}$$

CFG: Formal Definition (2): Example

- Design the CFG for strings of properly-nested parentheses.
 e.g., $()$, $()()$, $((()()))()$, *etc.*

Present your answer in a formal manner.

- $G = (\{S\}, \{(\,)\}, R, S)$, where R is

$$S \rightarrow (S) \mid SS \mid \epsilon$$

- Draw *parse trees* for the above three strings that G generates.

CFG: Formal Definition (3): Example

- Consider the grammar $G = (V, \Sigma, R, S)$:

- R is

$$\begin{array}{lcl}
 Expr & \rightarrow & Expr + Term \\
 & | & Term \\
 Term & \rightarrow & Term * Factor \\
 & | & Factor \\
 Factor & \rightarrow & (Expr) \\
 & | & a
 \end{array}$$

- $V = \{Expr, Term, Factor\}$
- $\Sigma = \{a, +, *, (,)\}$
- $S = Expr$
- Precedence** of operators $+$ and $*$ is embedded in the grammar.
 - “Plus” is specified at a **higher** level (*Expr*) than is “times” (*Term*).
 - Both operands of a multiplication (*Factor*) may be **parenthesized**.

Regular Expressions to CFG's

- Recall the semantics of regular expressions (assuming that we do not consider \emptyset):

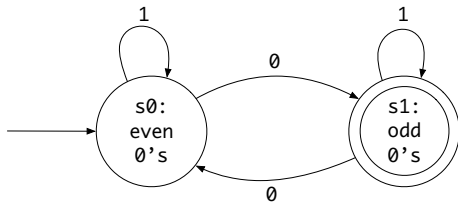
$$\begin{aligned}
 L(\epsilon) &= \{\epsilon\} \\
 L(a) &= \{a\} \\
 L(E + F) &= L(E) \cup L(F) \\
 L(EF) &= L(E)L(F) \\
 L(E^*) &= (L(E))^* \\
 L(E) &= L(E)
 \end{aligned}$$

- e.g., Grammar for $(00 + 1)^* + (11 + 0)^*$

$$\begin{aligned}
 S &\rightarrow A \mid B \\
 A &\rightarrow \epsilon \mid AC \\
 C &\rightarrow 00 \mid 1 \\
 B &\rightarrow \epsilon \mid BD \\
 D &\rightarrow 11 \mid 0
 \end{aligned}$$

DFA to CFG's

- Given a DFA $M = (Q, \Sigma, \delta, q_0, F)$:
 - Make a variable R_i for each state $q_i \in Q$.
 - Make R_0 the start variable, where q_0 is the start state of M .
 - Add a rule $R_i \rightarrow aR_j$ to the grammar if $\delta(q_i, a) = q_j$.
 - Add a rule $R_i \rightarrow \epsilon$ if $q_i \in F$.
- e.g., Grammar for



$$R_0 \rightarrow 1R_0 \mid 0R_1$$

$$R_1 \rightarrow 0R_0 \mid 1R_1 \mid \epsilon$$

CFG: Leftmost Derivations (1)

$$\begin{aligned} \text{Expr} &\rightarrow \text{Expr} + \text{Term} \mid \text{Term} \\ \text{Term} &\rightarrow \text{Term} * \text{Factor} \mid \text{Factor} \\ \text{Factor} &\rightarrow (\text{Expr}) \mid a \end{aligned}$$

- **Unique leftmost derivation** for the string $a + a * a$:

$$\begin{aligned} \text{Expr} &\Rightarrow \text{Expr} + \text{Term} \\ &\Rightarrow \text{Term} + \text{Term} \\ &\Rightarrow \text{Factor} + \text{Term} \\ &\Rightarrow a + \text{Term} \\ &\Rightarrow a + \text{Term} * \text{Factor} \\ &\Rightarrow a + \text{Factor} * \text{Factor} \\ &\Rightarrow a + a * \text{Factor} \\ &\Rightarrow a + a * a \end{aligned}$$

- This **leftmost derivation** suggests that $a * a$ is the right operand of $+$.

CFG: Rightmost Derivations (1)

$$\begin{aligned} \text{Expr} &\rightarrow \text{Expr} + \text{Term} \mid \text{Term} \\ \text{Term} &\rightarrow \text{Term} * \text{Factor} \mid \text{Factor} \\ \text{Factor} &\rightarrow (\text{Expr}) \mid a \end{aligned}$$

- **Unique rightmost derivation** for the string $a + a * a$:

$$\begin{aligned} \text{Expr} &\Rightarrow \text{Expr} + \text{Term} \\ &\Rightarrow \text{Expr} + \text{Term} * \text{Factor} \\ &\Rightarrow \text{Expr} + \text{Term} * a \\ &\Rightarrow \text{Expr} + \text{Factor} * a \\ &\Rightarrow \text{Expr} + a * a \\ &\Rightarrow \text{Term} + a * a \\ &\Rightarrow \text{Factor} + a * a \\ &\Rightarrow a + a * a \end{aligned}$$

- This **rightmost derivation** suggests that $a * a$ is the right operand of $+$.

CFG: Leftmost Derivations (2)

$$\text{Expr} \rightarrow \text{Expr} + \text{Term} \mid \text{Term}$$

$$\text{Term} \rightarrow \text{Term} * \text{Factor} \mid \text{Factor}$$

$$\text{Factor} \rightarrow (\text{Expr}) \mid a$$

- **Unique leftmost derivation** for the string $(a + a) * a$:

$$\text{Expr} \Rightarrow \text{Term}$$

$$\Rightarrow \text{Term} * \text{Factor}$$

$$\Rightarrow \text{Factor} * \text{Factor}$$

$$\Rightarrow (\text{Expr}) * \text{Factor}$$

$$\Rightarrow (\text{Expr} + \text{Term}) * \text{Factor}$$

$$\Rightarrow (\text{Term} + \text{Term}) * \text{Factor}$$

$$\Rightarrow (\text{Factor} + \text{Term}) * \text{Factor}$$

$$\Rightarrow (a + \text{Term}) * \text{Factor}$$

$$\Rightarrow (a + \text{Factor}) * \text{Factor}$$

$$\Rightarrow (a + a) * \text{Factor}$$

$$\Rightarrow (a + a) * a$$

This **leftmost derivation** suggests that $(a + a)$ is the left

CFG: Rightmost Derivations (2)

$$\begin{aligned} \text{Expr} &\rightarrow \text{Expr} + \text{Term} \mid \text{Term} \\ \text{Term} &\rightarrow \text{Term} * \text{Factor} \mid \text{Factor} \\ \text{Factor} &\rightarrow (\text{Expr}) \mid a \end{aligned}$$

- **Unique rightmost derivation** for the string $(a + a) * a$:

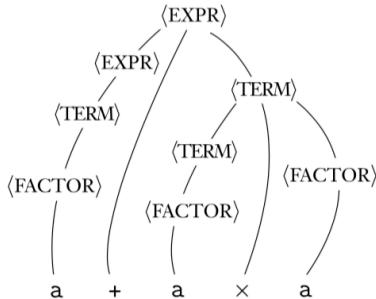
$$\begin{aligned} \text{Expr} &\Rightarrow \text{Term} \\ &\Rightarrow \text{Term} * \text{Factor} \\ &\Rightarrow \text{Term} * a \\ &\Rightarrow \text{Factor} * a \\ &\Rightarrow (\text{Expr}) * a \\ &\Rightarrow (\text{Expr} + \text{Term}) * a \\ &\Rightarrow (\text{Expr} + \text{Factor}) * a \\ &\Rightarrow (\text{Expr} + a) * a \\ &\Rightarrow (\text{Term} + a) * a \\ &\Rightarrow (\text{Factor} + a) * a \\ &\Rightarrow (a + a) * a \end{aligned}$$

This **rightmost derivation** suggests that $(a + a)$ is the left operand of $*$.

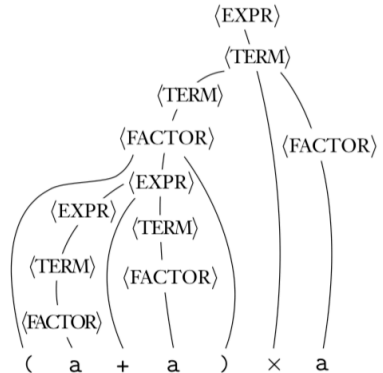
CFG: Parse Trees vs. Derivations (1)

- Parse trees for (leftmost & rightmost) derivations of expressions:

$a + a * a$



$(a + a) * a$



- Orders in which derivations are performed are **not** reflected on parse trees.

CFG: Parse Trees vs. Derivations (2)

- A string $w \in \Sigma^*$ may have more than one *derivations*.
Q: distinct *derivations* for $w \in \Sigma^*$ \Rightarrow distinct *parse trees* for w ?
A: Not in general \because Derivations with **distinct orders** of variable substitutions may still result in the **same parse tree**.
- For example:

$$\begin{aligned}
 \text{Expr} &\rightarrow \text{Expr} + \text{Term} \mid \text{Term} \\
 \text{Term} &\rightarrow \text{Term} * \text{Factor} \mid \text{Factor} \\
 \text{Factor} &\rightarrow (\text{Expr}) \mid a
 \end{aligned}$$

For string $a + a * a$, the **leftmost** and **rightmost** derivations have **distinct orders** of variable substitutions, but their corresponding **parse trees are the same**.

CFG: Ambiguity: Definition

Given a grammar $G = (V, \Sigma, R, S)$:

- A string $w \in \Sigma^*$ is derived **ambiguously** in G if there exist two or more **distinct parse trees** or, equally, two or more **distinct leftmost** derivations or, equally, two or more **distinct rightmost** derivations.

Here we require that all such derivations have been completed by following a particular order (leftmost or rightmost) to avoid **false alarm**.

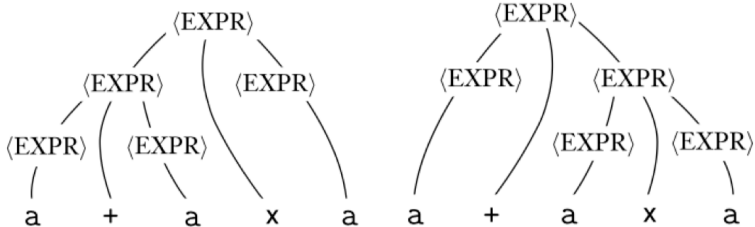
- G is **ambiguous** if it generates some string ambiguously.

CFG: Ambiguity: Exercise (1)

- Is the following grammar *ambiguous*?

$$\text{Expr} \rightarrow \text{Expr} + \text{Expr} \mid \text{Expr} * \text{Expr} \mid (\text{Expr}) \mid a$$

- Yes \because it generates the string $a + a * a$ *ambiguously*:



- Distinct ASTs** (for the **same input**) mean **distinct semantic interpretations**: e.g.,
when a post-order traversal is used to implement evaluation
- Exercise:** Show *leftmost* derivations for the two parse trees.

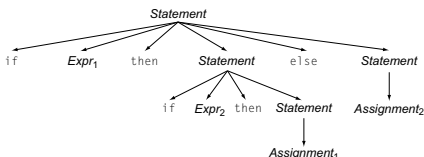
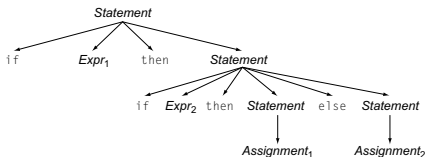
CFG: Ambiguity: Exercise (2.1)

- Is the following grammar **ambiguous**?

$$\begin{array}{l}
 \text{Statement} \rightarrow \text{if } \text{Expr} \text{ then } \text{Statement} \\
 \quad \quad \quad | \text{if } \text{Expr} \text{ then } \text{Statement} \text{ else } \text{Statement} \\
 \quad \quad \quad | \text{Assignment} \\
 \quad \quad \quad \dots
 \end{array}$$

- Yes \because it generates the following string **ambiguously**:

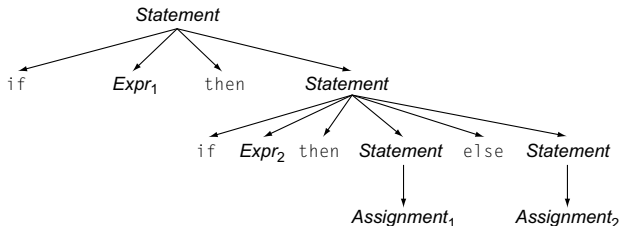
if Expr_1 then if Expr_2 then Assignment_1 else Assignment_2



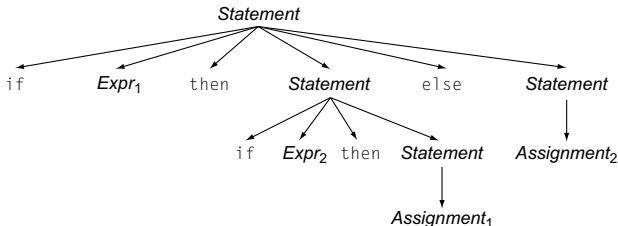
- This is called the **dangling else** problem.
- Exercise:** Show **leftmost** derivations for the two parse trees.

CFG: Ambiguity: Exercise (2.2)

(**Meaning 1**) $Assignment_2$ may be associated with the inner `if`:



(**Meaning 2**) $Assignment_2$ may be associated with the outer `if`:



CFG: Ambiguity: Exercise (2.3)

- We may remove the **ambiguity** by specifying that the **dangling else** is associated with the **nearest if**:

```

Statement  →  if Expr then Statement
              |  if Expr then WithElse else Statement
              |  Assignment
WithElse    →  if Expr then WithElse else WithElse
              |  Assignment
  
```

- When applying `if ... then WithElse else Statement`:
 - The **true** branch will be produced via **WithElse**.
 - The **false** branch will be produced via **Statement**.

There is **no circularity** between the two non-terminals.

Discovering Derivations

- Given a CFG $G = (V, \Sigma, R, S)$ and an input program $p \in \Sigma^*$:
 - So far we **manually** come up a valid derivation $S \xrightarrow{*} p$.
 - A parser is supposed to **automate** this derivation process.
 - Given an input sequence of (t, c) pairs, where **token** t (e.g., r241) belongs to some **syntactic category** c (e.g., register):
 - Either output a **valid derivation** (as an **AST**), or signal an **error**.
- In the process of building an **AST** for the input program:
 - Root** of AST: **start symbol** S of G
 - Internal nodes**: A subset of variables V of G
 - Leaves** of AST: **token sequence** input by the scanner
 - \Rightarrow Discovering the **grammatical connections** (according to R) between the root, internal nodes, and leaves is the hard part!
- Approaches to Parsing: $[w \in (V \cup \Sigma)^*, A \in V, \boxed{A \rightarrow w} \in R]$
 - Top-down** parsing
 - For a node representing A , extend it with a subtree representing w .
 - Bottom-up** parsing
 - For a substring matching w , build a node representing A accordingly.

TDP: Discovering Leftmost Derivation

```
ALGORITHM: TDParse
INPUT: CFG  $G = (V, \Sigma, R, S)$ 
OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus  $\in V$  then
      if  $\exists$  unvisited rule  $focus \rightarrow \beta_1\beta_2\dots\beta_n \in R$  then
        create  $\beta_1, \beta_2\dots\beta_n$  as children of focus
        trace.push( $\beta_n\beta_{n-1}\dots\beta_2$ )
        focus :=  $\beta_1$ 
      else
        if focus = S then report syntax error
        else backtrack
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF  $\wedge$  focus = null then return root
    else backtrack
```

backtrack \triangleq pop *focus.siblings*; *focus* := *focus.parent*; *focus.resetChildren*

TDP: Exercise (1)

- Given the following CFG **G**:

$$\begin{array}{lcl}
 \textit{Expr} & \rightarrow & \textit{Expr} + \textit{Term} \\
 & | & \textit{Term} \\
 \textit{Term} & \rightarrow & \textit{Term} * \textit{Factor} \\
 & | & \textit{Factor} \\
 \textit{Factor} & \rightarrow & (\textit{Expr}) \\
 & | & a
 \end{array}$$

Trace *TDParse* on how to build an AST for input $a + a * a$.

- Running *TDParse* with **G** results an **infinite loop** !!!
 - TDParse* focuses on the **leftmost** non-terminal.
 - The grammar **G** contains **left-recursions**.
- We must first convert left-recursions in **G** to **right-recursions**.

TDP: Exercise (2)

- Given the following CFG **G**:

$$\begin{array}{lcl}
 \text{Expr} & \rightarrow & \text{Term Expr}' \\
 \text{Expr}' & \rightarrow & + \text{Term Expr}' \\
 & | & \epsilon \\
 \text{Term} & \rightarrow & \text{Factor Term}' \\
 \text{Term}' & \rightarrow & * \text{Factor Term}' \\
 & | & \epsilon \\
 \text{Factor} & \rightarrow & (\text{Expr}) \\
 & | & a
 \end{array}$$

Exercise. Trace *TDParse* on building AST for $a + a * a$.

Exercise. Trace *TDParse* on building AST for $(a + a) * a$.

Q: How to handle ϵ -productions (e.g., $\text{Expr} \rightarrow \epsilon$)?

A: Execute *focus* := *trace.pop()* to advance to next node.

- Running *TDParse* will **terminate** \because **G** is **right-recursive**.
- We will learn about a systematic approach to converting left-recursions in a given grammar to **right-recursions**.

Left-Recursions (LR): Direct vs. Indirect

Given CFG $G = (V, \Sigma, R, S)$, $\alpha, \beta, \gamma \in (V \cup \Sigma)^*$, G contains:

- A **cycle** if $\exists A \in V \bullet A \xRightarrow{*} A$
- A **direct** LR if $A \rightarrow A\alpha \in R$ for non-terminal $A \in V$

e.g.,

$Expr$	\rightarrow	$Expr + Term$
		$Term$
$Term$	\rightarrow	$Term * Factor$
		$Factor$
$Factor$	\rightarrow	$(Expr)$
		a

e.g.,

$Expr$	\rightarrow	$Expr + Term$
		$Expr - Term$
		$Term$
$Term$	\rightarrow	$Term * Factor$
		$Term / Factor$
		$Factor$

- An **indirect** LR if $A \rightarrow B\beta \in R$ for non-terminals $A, B \in V$, $B \xRightarrow{*} A\gamma$

A	\rightarrow	Br
B	\rightarrow	Cd
C	\rightarrow	At

$A \rightarrow Br, B \xRightarrow{*} Atd$

A	\rightarrow	Ba		b
B	\rightarrow	Cd		e
C	\rightarrow	Df		g
D	\rightarrow	f		$Aa \quad \quad Cg$

$A \rightarrow Ba, B \xRightarrow{*} Aafd$

TDP: (Preventively) Eliminating LRs

```

1  ALGORITHM: RemoveLR
2  INPUT: CFG  $G = (V, \Sigma, R, S)$ 
3  ASSUME:  $G$  acyclic  $\wedge$  with no  $\epsilon$ -productions
4  OUTPUT:  $G'$  s.t.  $G' \equiv G$ ,  $G'$  has no
5           indirect & direct left-recursions
6  PROCEDURE:
7     impose an order on  $V$ :  $\langle\langle A_1, A_2, \dots, A_n \rangle\rangle$ 
8     for  $i$ : 1 ..  $n$ :
9         for  $j$ : 1 ..  $i-1$ :
10            if  $\exists A_j \rightarrow A_j \gamma \in R \wedge A_j \rightarrow \delta_1 | \delta_2 | \dots | \delta_m \in R$  then
11                replace  $A_j \rightarrow A_j \gamma$  with  $A_j \rightarrow \delta_1 \gamma | \delta_2 \gamma | \dots | \delta_m \gamma$ 
12            end
13        for  $A_i \rightarrow A_i \alpha | \beta \in R$ :
14            replace it with:  $A_i \rightarrow \beta A', A' \rightarrow \alpha A' | \epsilon$ 

```

L9 to L11: Remove *indirect* left-recursions from A_1 to A_{i-1} .

L12 to L13: Remove *direct* left-recursions from A_1 to A_{i-1} .

Loop Invariant (outer for-loop)? At the start of i^{th} iteration:

- No direct or indirect left-recursions for A_1, A_2, \dots, A_{i-1} .
- More precisely: $\forall k : k < i \bullet \neg(\exists l \bullet l \leq k \wedge A_k \rightarrow A_l \dots \in R)$

CFG: Eliminating ϵ -Productions (1)

- Motivations:
 - *TDParse* requires CFG with no ϵ -productions.
 - *RemoveLR* produces CFG which may contain ϵ -productions.
- $\epsilon \notin L \Rightarrow \exists$ CFG $G = (V, \Sigma, R, S)$ s.t. G has no ϵ -productions.
 - An ϵ -production has the form $A \rightarrow \epsilon$.
- A variable A is **nullable** if $A \xRightarrow{*} \epsilon$.
 - Each terminal symbol is *not nullable*.
 - Variable A is **nullable** if either:
 - $A \rightarrow \epsilon \in R$; or
 - $A \rightarrow B_1 B_2 \dots B_k \in R$, where each variable B_i ($1 \leq i \leq k$) is a **nullable**.
- Given a production $B \rightarrow CAD$, if only variable A is nullable, then there are 2 versions of B : $B \rightarrow CAD \mid CD$
- In general, given a production $A \rightarrow X_1 X_2 \dots X_k$ with k symbols, if m of the k symbols are **nullable**:
 - $m < k$: There are 2^m versions of A .
 - $m = k$: There are $2^m - 1$ versions of A . [excluding $A \rightarrow \epsilon$]

CFG: Eliminating ϵ -Productions (2)

- Eliminate ϵ -productions from the following grammar:

$$\begin{aligned}
 S &\rightarrow AB \\
 A &\rightarrow aAA \mid \epsilon \\
 B &\rightarrow bBB \mid \epsilon
 \end{aligned}$$

- Which are the *nullable* variables?

[S, A, B]

$$\begin{aligned}
 S &\rightarrow A \mid B \mid AB && \{S \rightarrow \epsilon \text{ not included}\} \\
 A &\rightarrow aAA \mid aA \mid a && \{A \rightarrow aA \text{ duplicated}\} \\
 B &\rightarrow bBB \mid bB \mid b && \{B \rightarrow bB \text{ duplicated}\}
 \end{aligned}$$

Backtrack-Free Parsing (1)

- TDParse automates the *top-down, leftmost* derivation process by consistently choosing production rules (e.g., in order of their appearance in CFG).
 - This *inflexibility* may lead to *inefficient* runtime performance due to the need to *backtrack*.
 - e.g., It may take the *construction of a giant subtree* to find out a *mismatch* with the input tokens, which end up requiring it to *backtrack* all the way back to the *root* (start symbol).
- We may avoid backtracking with a modification to the parser:
 - When deciding which production rule to choose, consider:
 - (1) the *current* input symbol
 - (2) the *consequential first* symbol if a rule was applied for *focus* [*lookahead* symbol]
 - Using a *one symbol lookahead*, w.r.t. a *right-recursive* CFG, each alternative for the *leftmost nonterminal* leads to a *unique terminal*, allowing the parser to decide on a choice that prevents *backtracking*.
 - Such CFG is *backtrack free* with a *lookahead* of one symbol.
 - We also call such backtrack-free CFG a *predictive grammar*.

The FIRST Set: Definition

- Say we write $T \subset \mathbb{P}(\Sigma^*)$ to denote the set of valid tokens recognizable by the scanner.
- **FIRST** $(\alpha) \triangleq$ set of symbols that can appear as the *first word* in some string derived from α .
- More precisely:

$$\text{FIRST}(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha \in T \\ \{w \mid w \in \Sigma^* \wedge \alpha \xRightarrow{*} w\beta \wedge \beta \in (V \cup \Sigma)^*\} & \text{if } \alpha \in V \end{cases}$$

The FIRST Set: Examples

- Consider this *right*-recursive CFG:

0	<i>Goal</i>	\rightarrow	<i>Expr</i>	6	<i>Term'</i>	\rightarrow	\times <i>Factor</i> <i>Term'</i>
1	<i>Expr</i>	\rightarrow	<i>Term Expr'</i>	7			\div <i>Factor</i> <i>Term'</i>
2	<i>Expr'</i>	\rightarrow	$+$ <i>Term</i> <i>Expr'</i>	8			ϵ
3			$-$ <i>Term</i> <i>Expr'</i>	9	<i>Factor</i>	\rightarrow	$($ <i>Expr</i> $)$
4			ϵ	10			num
5	<i>Term</i>	\rightarrow	<i>Factor</i> <i>Term'</i>	11			name

- Compute **FIRST** for each terminal (e.g., num, +, ()):

	num	name	+	-	\times	\div	$($	$)$	eof	ϵ
FIRST	num	name	+	-	\times	\div	$($	$)$	eof	ϵ

- Compute **FIRST** for each non-terminal (e.g., *Expr*, *Term'*):

	<i>Expr</i>	<i>Expr'</i>	<i>Term</i>	<i>Term'</i>	<i>Factor</i>
FIRST	$($, name, num	$+$, $-$, ϵ	$($, name, num	\times , \div , ϵ	$($, name, num

Computing the FIRST Set

$$\text{FIRST}(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha \in T \\ \{w \mid w \in \Sigma^* \wedge \alpha \xRightarrow{*} w\beta \wedge \beta \in (V \cup \Sigma)^*\} & \text{if } \alpha \in V \end{cases}$$

ALGORITHM: *GetFirst*

INPUT: CFG $G = (V, \Sigma, R, S)$

$T \subset \Sigma^*$ denotes valid terminals

OUTPUT: $\text{FIRST} : V \cup T \cup \{\epsilon, eof\} \rightarrow \mathbb{P}(T \cup \{\epsilon, eof\})$

PROCEDURE:

for $\alpha \in (T \cup \{eof, \epsilon\})$: $\text{FIRST}(\alpha) := \{\alpha\}$

for $A \in V$: $\text{FIRST}(A) := \emptyset$

$lastFirst := \emptyset$

while ($lastFirst \neq \text{FIRST}$):

$lastFirst := \text{FIRST}$

for $A \rightarrow \beta_1\beta_2\dots\beta_k \in R$ s.t. $\forall \beta_j: \beta_j \in (T \cup V)$:

$rhs := \text{FIRST}(\beta_1) - \{\epsilon\}$

for ($i := 1$; $\epsilon \in \text{FIRST}(\beta_i) \wedge i < k$; $i++$):

$rhs := rhs \cup (\text{FIRST}(\beta_{i+1}) - \{\epsilon\})$

if $i = k \wedge \epsilon \in \text{FIRST}(\beta_k)$ **then**

$rhs := rhs \cup \{\epsilon\}$

end

$\text{FIRST}(A) := \text{FIRST}(A) \cup rhs$

Computing the FIRST Set: Extension

- Recall: **FIRST** takes as input a token or a variable.

$$\mathbf{FIRST} : V \cup T \cup \{\epsilon, eof\} \rightarrow \mathbb{P}(T \cup \{\epsilon, eof\})$$

- The computation of variable *rhs* in algorithm `GetFirst` actually suggests an extended, overloaded version:

$$\mathbf{FIRST} : (V \cup T \cup \{\epsilon, eof\})^* \rightarrow \mathbb{P}(T \cup \{\epsilon, eof\})$$

FIRST may also take as input a string $\beta_1\beta_2\dots\beta_n$ (RHS of rules).

- More precisely:

$$\mathbf{FIRST}(\beta_1\beta_2\dots\beta_n) =$$

$$\left\{ \begin{array}{l} \mathbf{FIRST}(\beta_1) \cup \mathbf{FIRST}(\beta_2) \cup \dots \cup \mathbf{FIRST}(\beta_k) \\ \wedge \\ \epsilon \notin \mathbf{FIRST}(\beta_k) \end{array} \middle| \begin{array}{l} \forall i: 1 \leq i < k \bullet \epsilon \in \mathbf{FIRST}(\beta_i) \end{array} \right\}$$

Note. β_k is the first symbol whose **FIRST** set does not contain ϵ .

Extended FIRST Set: Examples

Consider this *right*-recursive CFG:

0	$Goal \rightarrow Expr$	6	$Term' \rightarrow \times Factor Term'$
1	$Expr \rightarrow Term Expr'$	7	$ \div Factor Term'$
2	$Expr' \rightarrow + Term Expr'$	8	$ \epsilon$
3	$ - Term Expr'$	9	$Factor \rightarrow (Expr)$
4	$ \epsilon$	10	$ num$
5	$Term \rightarrow Factor Term'$	11	$ name$

e.g., $FIRST(Term Expr') = FIRST(Term) = \{ (, name, num \}$

e.g., $FIRST(+ Term Expr') = FIRST(+) = \{ + \}$

e.g., $FIRST(- Term Expr') = FIRST(-) = \{ - \}$

e.g., $FIRST(\epsilon) = \{ \epsilon \}$

Is the FIRST Set Sufficient

- Consider the following three productions:

$Expr'$	\rightarrow	$+$	$Term$	$Term'$	(1)
		$-$	$Term$	$Term'$	(2)
		ϵ			(3)

In TDP, when the parser attempts to expand an $Expr'$ node, it **looks ahead with one symbol** to decide on the choice of rule:

$\mathbf{FIRST}(+) = \{+\}$, $\mathbf{FIRST}(-) = \{-\}$, and $\mathbf{FIRST}(\epsilon) = \{\epsilon\}$.

Q. When to choose rule (3) (causing **focus := trace.pop()**)?

A? Choose rule (3) when $focus \neq \mathbf{FIRST}(+) \wedge focus \neq \mathbf{FIRST}(-)$?

- Correct** but **inefficient** in case of illegal input string: syntax error is only reported after possibly a long series of **backtrack**.
- Useful if parser knows which words can appear, after an application of the ϵ -production (rule (3)), as leading symbols.
- FOLLOW** ($v : V$) \triangleq set of symbols that can appear to the immediate right of a string derived from α .

$$\mathbf{FOLLOW}(v) = \{w \mid w, x, y \in \Sigma^* \wedge v \xRightarrow{*} x \wedge S \xRightarrow{*} xwy\}$$

The FOLLOW Set: Examples

- Consider this *right*-recursive CFG:

0	<i>Goal</i>	\rightarrow	<i>Expr</i>	6	<i>Term'</i>	\rightarrow	\times <i>Factor</i> <i>Term'</i>
1	<i>Expr</i>	\rightarrow	<i>Term Expr'</i>	7			\div <i>Factor</i> <i>Term'</i>
2	<i>Expr'</i>	\rightarrow	$+$ <i>Term Expr'</i>	8			ϵ
3			$-$ <i>Term Expr'</i>	9	<i>Factor</i>	\rightarrow	$($ <i>Expr</i> $)$
4			ϵ	10			num
5	<i>Term</i>	\rightarrow	<i>Factor Term'</i>	11			name

- Compute **FOLLOW** for each non-terminal (e.g., *Expr*, *Term'*):

	<i>Expr</i>	<i>Expr'</i>	<i>Term</i>	<i>Term'</i>	<i>Factor</i>
FOLLOW	eof, $)$	eof, $)$	eof, +, -, $)$	eof, +, -, $)$	eof, +, -, x, \div , $)$

Computing the FOLLOW Set

$$\text{FOLLOW}(v) = \{w \mid w, x, y \in \Sigma^* \wedge v \xrightarrow{*} x \wedge S \xrightarrow{*} xwy\}$$

ALGORITHM: *GetFollow*

INPUT: CFG $G = (V, \Sigma, R, S)$

OUTPUT: FOLLOW: $V \rightarrow \mathbb{P}(T \cup \{\text{eof}\})$

PROCEDURE:

for $A \in V$: FOLLOW(A) := \emptyset

FOLLOW(S) := {eof}

lastFollow := \emptyset

while (*lastFollow* \neq FOLLOW):

lastFollow := FOLLOW

for $A \rightarrow \beta_1\beta_2\dots\beta_k \in R$:

A)

for $i: k \dots 1$:

if $\beta_i \in V$ **then**

 FOLLOW(β_i) := FOLLOW(β_i) \cup trailer

if $\epsilon \in \text{FIRST}(\beta_i)$

then trailer := trailer \cup (FIRST(β_i) $- \epsilon$)

else trailer := FIRST(β_i)

else

 trailer := FIRST(β_i)

Backtrack-Free Grammar

- A **backtrack-free grammar** (for a **top-down parser**), when expanding the **focus internal node**, is always able to choose a **unique** rule with the **one-symbol lookahead** (or report a **syntax error** when no rule applies).
- To formulate this, we first define:

$$\mathbf{FIRST}^+(A \rightarrow \beta) = \begin{cases} \mathbf{FIRST}(\beta) & \text{if } \epsilon \notin \mathbf{FIRST}(\beta) \\ \mathbf{FIRST}(\beta) \cup \mathbf{FOLLOW}(A) & \text{otherwise} \end{cases}$$

$\mathbf{FIRST}(\beta)$ is the extended version where β may be $\beta_1\beta_2 \dots \beta_n$

- Now, a **backtrack-free grammar** has each of its productions $A \rightarrow \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_n$ satisfying:

$$\forall i, j: 1 \leq i, j \leq n \wedge i \neq j \bullet \mathbf{FIRST}^+(\gamma_i) \cap \mathbf{FIRST}^+(\gamma_j) = \emptyset$$

TDP: Lookahead with One Symbol

```
ALGORITHM: TDParse
INPUT: CFG  $G = (V, \Sigma, R, S)$ 
OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol  $S$ 
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus  $\in V$  then % use FOLLOW set as well?
      if  $\exists$  unvisited rule focus  $\rightarrow \beta_1\beta_2\dots\beta_n \in R \wedge$  word  $\in \text{FIRST}^+(\beta)$  then
        create  $\beta_1, \beta_2, \dots, \beta_n$  as children of focus
        trace.push( $\beta_n\beta_{n-1}\dots\beta_2$ )
        focus :=  $\beta_1$ 
      else
        if focus =  $S$  then report syntax error
        else backtrack
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF  $\wedge$  focus = null then return root
    else backtrack
```

backtrack \triangleq pop *focus*.siblings; *focus* := *focus*.parent; *focus*.resetChildren

Backtrack-Free Grammar: Exercise

Is the following CFG *backtrack free*?

11	<i>Factor</i>	→	name
12			name [<i>ArgList</i>]
13			name (<i>ArgList</i>)
15	<i>ArgList</i>	→	<i>Expr</i> <i>MoreArgs</i>
16	<i>MoreArgs</i>	→	, <i>Expr</i> <i>MoreArgs</i>
17			ε

- $\epsilon \notin \mathbf{FIRST}(Factor) \Rightarrow \mathbf{FIRST}^+(Factor) = \mathbf{FIRST}(Factor)$
- $\mathbf{FIRST}(Factor \rightarrow \text{name}) = \{\text{name}\}$
- $\mathbf{FIRST}(Factor \rightarrow \text{name } [ArgList]) = \{\text{name}\}$
- $\mathbf{FIRST}(Factor \rightarrow \text{name } (ArgList)) = \{\text{name}\}$

∴ The above grammar is *not* backtrack free.

⇒ To expand an AST node of *Factor*, with a *lookahead* of name, the parser has no basis to choose among rules 11, 12, and 13.

Backtrack-Free Grammar: Left-Factoring

- A CFG is not backtrack free if there exists a **common prefix** (name) among the RHS of **multiple** production rules.
- To make such a CFG **backtrack-free**, we may transform it using **left factoring**: a process of extracting and isolating **common prefixes** in a set of production rules.

- Identify a common prefix α :

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_j$$

[each of $\gamma_1, \gamma_2, \dots, \gamma_j$ does not begin with α]

- Rewrite that production rule as:

$$A \rightarrow \alpha B \mid \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_j$$

$$B \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

- New rule $B \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$ may also contain **common prefixes**.
- Rewriting **continues** until no common prefixes are identified.

Left-Factoring: Exercise

- Use **left-factoring** to remove all **common prefixes** from the following grammar.

11	<i>Factor</i>	→	name
12			name [<i>ArgList</i>]
13			name (<i>ArgList</i>)
15	<i>ArgList</i>	→	<i>Expr MoreArgs</i>
16	<i>MoreArgs</i>	→	, <i>Expr MoreArgs</i>
17			ε

- Identify common prefix name and rewrite rules 11, 12, and 13:

<i>Factor</i>	→	name	<i>Arguments</i>		
<i>Arguments</i>	→	[<i>ArgList</i>]	
			(<i>ArgList</i>)
			ε		

Any more **common prefixes**?

[No]

TDP: Terminating and Backtrack-Free

- Given an arbitrary CFG as input to a *top-down parser*:
 - Q. How do we avoid a *non-terminating* parsing process?
 - A. Convert left-recursions to right-recursion.
 - Q. How do we minimize the need of *backtracking*?
 - A. left-factoring & one-symbol lookahead using **FIRST⁺**
- Not** every context-free *language* has a corresponding *backtrack-free* context-free *grammar*.

Given a CFL I , the following is **undecidable**:

$$\exists \text{cfg} \mid L(\text{cfg}) = I \wedge \text{isBacktrackFree}(\text{cfg})$$

- Given a CFG $g = (V, \Sigma, R, S)$, whether or not g is *backtrack-free* is **decidable**:

For each $A \rightarrow \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_n \in R$:

$$\forall i, j : 1 \leq i, j \leq n \wedge i \neq j \bullet \text{FIRST}^+(\gamma_i) \cap \text{FIRST}^+(\gamma_j) = \emptyset$$

Backtrack-Free Parsing (2.1)

- A **recursive-descent** parser is:
 - A top-down parser
 - Structured as a set of **mutually recursive** procedures
 - Each procedure corresponds to a **non-terminal** in the grammar.
 - See an example.
- Given a **backtrack-free** grammar, a tool (a.k.a. **parser generator**) can automatically generate:
 - **FIRST**, **FOLLOW**, and **FIRST⁺** sets
 - An efficient **recursive-descent** parser
 - This generated parser is called an **LL(1) parser**, which:
 - Processes input from Left to right
 - Constructs a Leftmost derivation
 - Uses a lookahead of 1 symbol
- **LL(1) grammars** are those working in an **LL(1)** scheme.
 - LL(1) grammars** are **backtrack-free** by definition.

Backtrack-Free Parsing (2.2)

- Consider this CFG with FIRST^+ sets of the RHSs:

	Production	FIRST^+
2	$\text{Expr}' \rightarrow + \text{Term Expr}'$	{+}
3	$\text{Expr}' \mid - \text{Term Expr}'$	{-}
4	$\text{Expr}' \mid \epsilon$	{ ϵ , eof, $_$ }

- The corresponding *recursive-descent* parser is structured as:

```

ExprPrim()
  if word = + ∨ word = - then /* Rules 2, 3 */
    word := NextWord()
    if(Term())
      then return ExprPrim()
      else return false
  elseif word = ) ∨ word = eof then /* Rule 4 */
    return true
  else
    report a syntax error
    return false
  end

Term()
  ...
  
```

See: parser generator

LL(1) Parser: Exercise

Consider the following grammar:

$L \rightarrow R a$	$R \rightarrow aba$	$Q \rightarrow bbc$
$Q ba$	$caba$	bc
	$R bc$	

Q. Is it suitable for a *top-down predictive* parser?

- If so, show that it satisfies the **LL(1)** condition.
- If not, identify the problem(s) and correct it (them). Also show that the revised grammar satisfies the **LL(1)** condition.

BUP: Discovering Rightmost Derivation

- In TDP, we build the start variable as the **root node**, and then work towards the **leaves**. [**leftmost** derivation]
 - In Bottom-Up Parsing (BUP):
 - Words (terminals) are still returned from **left** to **right** by the scanner.
 - As terminals, or a mix of terminals and variables, are identified as **reducible** to some variable A (i.e., matching the RHS of some production rule for A), then a layer is added.
 - Eventually:
 - **accept**:
The **start variable** is reduced and **all** words have been consumed.
 - **reject**:
The next word is not ϵ or f , but no further **reduction** can be identified.
- Q.** Why can BUP find the **rightmost** derivation (RMD), if any?
- A.** BUP discovers steps in a **RMD** in its **reverse** order.

BUP: Discovering Rightmost Derivation (1)

- *table*-driven **LR(1)** parser: an implementation for BUP, which
 - Processes input from Left to right
 - Constructs a Rightmost derivation
 - Uses a lookahead of 1 symbol
- A language has the **LR(1)** property if it:
 - Can be parsed in a single Left to right scan,
 - To build a *reversed* Rightmost derivation,
 - Using a lookahead of 1 symbol to determine parsing actions.
- Critical step in a bottom-up parser is to find the *next* **handle**.

BUP: Discovering Rightmost Derivation (2)

```
ALGORITHM: BUParse
INPUT: CFG  $G = (V, \Sigma, R, S)$ , Action & Goto Tables
OUTPUT: Report Parse Success or Syntax Error
PROCEDURE:
  initialize an empty stack trace
  trace.push(S) /* start state */
  word := NextWord()
  while (true)
    state := trace.top()
    act := Action[state, word]
    if act = ``accept'' then
      succeed()
    elseif act = ``reduce  $A \rightarrow \beta$ '' then
      trace.pop()  $2 \times |\beta|$  times /* word + state */
      state := trace.top()
      trace.push(A)
      next := Goto[state, A]
      trace.push(next)
    elseif act = ``shift  $s_j$ '' then
      trace.push(word)
      trace.push( $s_j$ )
      word := NextWord()
    else
      fail()
```

BUP: Example Tracing (1)

- Consider the following grammar for parentheses:

1	$Goal \rightarrow List$
2	$List \rightarrow List Pair$
3	$\quad \quad Pair$
4	$Pair \rightarrow (Pair)$
5	$\quad \quad (\quad)$

- Assume: tables **Action** and **Goto** constructed accordingly:

State	Action Table			Goto Table	
	eof	()	List	Pair
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 6	s 7		5
4	r 2	r 2			
5			s 8		
6		s 6	s 10		9
7	r 5	r 5			
8	r 4	r 4			
9			s 11		
10			r 5		
11			r 4		

In **Action** table:

- s_i : shift to state i
- r_j : reduce to the LHS of production $\#j$

BUP: Example Tracing (2.1)

Consider the steps of performing BUP on input ():

Iteration	State	word	Stack	Handle	Action
<i>initial</i>	—	(\$ 0	— <i>none</i> —	—
1	0	(\$ 0	— <i>none</i> —	<i>shift 3</i>
2	3)	\$ 0 (3	— <i>none</i> —	<i>shift 7</i>
3	7	eof	\$ 0 (3) 7	()	<i>reduce 5</i>
4	2	eof	\$ 0 <i>Pair 2</i>	<i>Pair</i>	<i>reduce 3</i>
5	1	eof	\$ 0 <i>List 1</i>	<i>List</i>	<i>accept</i>

BUP: Example Tracing (2.2)

Consider the steps of performing BUP on input $(()) ()$:

Iteration	State	word	Stack	Handle	Action
<i>initial</i>	—	(\$ 0	— none —	—
1	0	(\$ 0	— none —	shift 3
2	3	(\$ 0 (3	— none —	shift 6
3	6)	\$ 0 (3 (6	— none —	shift 10
4	10)	\$ 0 (3 (6) 10	()	reduce 5
5	5)	\$ 0 (3 Pair 5	— none —	shift 8
6	8	(\$ 0 (3 Pair 5) 8	(Pair)	reduce 4
7	2	(\$ 0 Pair 2	Pair	reduce 3
8	1	(\$ 0 List 1	— none —	shift 3
9	3)	\$ 0 List 1 (3	— none —	shift 7
10	7	eof	\$ 0 List 1 (3) 7	()	reduce 5
11	4	eof	\$ 0 List 1 Pair 4	List Pair	reduce 2
12	1	eof	\$ 0 List 1	List	accept

BUP: Example Tracing (2.3)

Consider the steps of performing BUP on input ():

Iteration	State	word	Stack	Handle	Action
<i>initial</i>	—	<u>(</u>	\$ 0	— none —	—
1	0	<u>(</u>	\$ 0	— none —	shift 3
2	3	<u>)</u>	\$ 0 <u>(</u> 3	— none —	shift 7
3	7	<u>)</u>	\$ 0 <u>(</u> 3 <u>)</u> 7	— none —	error

LR(1) Items: Definition

- In **LR(1)** parsing, **Action** and **Goto** tables encode legitimate ways (w.r.t. a grammar) for finding *handles* (for *reductions*).
- In a **table-driven LR(1)** parser, the table-construction algorithm represents each potential *handle* (for a *reduction*) with an **LR(1) item** e.g.,

$$[A \rightarrow \beta \bullet \gamma, a]$$

where:

- A production rule $A \rightarrow \beta\gamma$ is currently being applied.
- A placeholder, \bullet , indicates the position of the parser's **stack top**.
 - ✓ The parser's stack contains β ("left context").
 - ✓ γ is yet to be matched.

Remark. Upon matching $\beta\gamma$, if a matches the current word, then we "replace" $\beta\gamma$ (and their corresponding states) with A (and its corresponding state).
- A terminal symbol a serves as a **lookahead symbol**.

LR(1) Items: Scenarios

An **LR(1) item** can be:

1. POSSIBILITY

$$[A \rightarrow \bullet\beta\gamma, a]$$

- In the current parsing context, an A would be valid.
- \bullet represents the position of the parser's **stack top**
- Recognizing a β next would be one step towards discovering an A .

2. PARTIALLY COMPLETION

$$[A \rightarrow \beta\bullet\gamma, a]$$

- The parser has progressed from $[A \rightarrow \bullet\beta\gamma, a]$ by recognizing β .
- Recognizing a γ next would be one step towards discovering an A .

3. COMPLETION

$$[A \rightarrow \beta\gamma\bullet, a]$$

- Parser has progressed from $[A \rightarrow \bullet\beta\gamma, a]$ by recognizing $\beta\gamma$.
- $\beta\gamma$ found in a context where an A followed by a would be valid.
- If the current input word matches a , then:
 - Current **complet item** is a **handle**.
 - Parser can **reduce** $\beta\gamma$ to A (and replace $\beta\gamma$ with A in its stack).

LR(1) Items: Example (1.1)

Consider the following grammar for parentheses:

1	$Goal \rightarrow List$
2	$List \rightarrow List\ Pair$
3	$Pair$
4	$Pair \rightarrow \underline{(}\ Pair\ \underline{)}$
5	$\underline{(}\ \underline{)}$

Initial State: $[Goal \rightarrow \bullet List, eof]$

Desired Final State: $[Goal \rightarrow List\bullet, eof]$

Intermediate States: Subset Construction

Q. Derive all **LR(1) items** for the above grammar.

- $FOLLOW(List) = \{eof, (\}$ $FOLLOW(Pair) = \{eof, (,)\}$
- For each production $A \rightarrow \beta$, given $FOLLOW(A)$, **LR(1) items** are:

$$\begin{aligned} & \{ [A \rightarrow \bullet\beta\gamma, a] \mid a \in FOLLOW(A) \} \\ & \cup \\ & \{ [A \rightarrow \beta\bullet\gamma, a] \mid a \in FOLLOW(A) \} \\ & \cup \\ & \{ [A \rightarrow \beta\gamma\bullet, a] \mid a \in FOLLOW(A) \} \end{aligned}$$

LR(1) Items: Example (1.2)

Q. Given production $A \rightarrow \beta$ (e.g., $Pair \rightarrow (Pair)$), how many **LR(1) items** can be generated?

- o The current parsing progress (on matching the RHS) can be:
 1. $\bullet(Pair)$
 2. $(\bullet Pair)$
 3. $(Pair \bullet)$
 4. $(Pair) \bullet$
- o Lookahead symbol following $Pair$? $FOLLOW(Pair) = \{eof, (,)\}$
- o All possible **LR(1) items** related to $Pair \rightarrow (Pair)$?
 - ✓ $[\bullet(Pair), eof]$ $[\bullet(Pair), (]$ $[\bullet(Pair),)]$
 - ✓ $[(\bullet Pair), eof]$ $[(\bullet Pair), (]$ $[(\bullet Pair),)]$
 - ✓ $[(Pair \bullet), eof]$ $[(Pair \bullet), (]$ $[(Pair \bullet),)]$
 - ✓ $[(Pair) \bullet, eof]$ $[(Pair) \bullet, (]$ $[(Pair) \bullet,)]$

A. How many in general (in terms of A and β)?

$$\underbrace{|\beta| + 1} \quad \times \quad \underbrace{|FOLLOW(A)|}$$

possible positions of \bullet possible lookahead symbols

LR(1) Items: Example (1.3)

A. There are 33 *LR(1) items* in the parentheses grammar.

$[Goal \rightarrow \bullet List, eof]$

$[Goal \rightarrow List \bullet, eof]$

$[List \rightarrow \bullet List Pair, eof]$ $[List \rightarrow \bullet List Pair, (]$

$[List \rightarrow List \bullet Pair, eof]$ $[List \rightarrow List \bullet Pair, (]$

$[List \rightarrow List Pair \bullet, eof]$ $[List \rightarrow List Pair \bullet, (]$

$[List \rightarrow \bullet Pair, eof]$ $[List \rightarrow \bullet Pair, (]$

$[List \rightarrow Pair \bullet, eof]$ $[List \rightarrow Pair \bullet, (]$

$[Pair \rightarrow \bullet (Pair), eof]$ $[Pair \rightarrow \bullet (Pair),)]$ $[Pair \rightarrow \bullet (Pair), (]$

$[Pair \rightarrow (\bullet Pair), eof]$ $[Pair \rightarrow (\bullet Pair),)]$ $[Pair \rightarrow (\bullet Pair), (]$

$[Pair \rightarrow (Pair \bullet), eof]$ $[Pair \rightarrow (Pair \bullet),)]$ $[Pair \rightarrow (Pair \bullet), (]$

$[Pair \rightarrow (Pair) \bullet, eof]$ $[Pair \rightarrow (Pair) \bullet,)]$ $[Pair \rightarrow (Pair) \bullet, (]$

$[Pair \rightarrow \bullet (), eof]$ $[Pair \rightarrow \bullet (), (]$ $[Pair \rightarrow \bullet (),)]$

$[Pair \rightarrow (\bullet), eof]$ $[Pair \rightarrow (\bullet), (]$ $[Pair \rightarrow (\bullet),)]$

$[Pair \rightarrow () \bullet, eof]$ $[Pair \rightarrow () \bullet, (]$ $[Pair \rightarrow () \bullet,)]$

LR(1) Items: Example (2)

Consider the following grammar for expressions:

0	$Goal \rightarrow Expr$	6	$Term' \rightarrow \times Factor Term'$
1	$Expr \rightarrow Term Expr'$	7	$\mid \div Factor Term'$
2	$Expr' \rightarrow + Term Expr'$	8	$\mid \epsilon$
3	$\mid - Term Expr'$	9	$Factor \rightarrow (Expr)$
4	$\mid \epsilon$	10	$\mid num$
5	$Term \rightarrow Factor Term'$	11	$\mid name$

Q. Derive all **LR(1) items** for the above grammar.

Hints. First compute **FOLLOW** for each non-terminal:

	<i>Expr</i>	<i>Expr'</i>	<i>Term</i>	<i>Term'</i>	<i>Factor</i>
FOLLOW	eof, <u>)</u>	eof, <u>)</u>	eof, +, -, <u>)</u>	eof, +, -, <u>)</u>	eof, +, -, x, ÷, <u>)</u>

Tips. Ignore ϵ **production** such as $Expr' \rightarrow \epsilon$ since the **FOLLOW** sets already take them into consideration.

Canonical Collection (CC) vs. LR(1) items

1	$Goal \rightarrow List$
2	$List \rightarrow List\ Pair$
3	$Pair$
4	$Pair \rightarrow (Pair)$
5	$()$

Recall:

LR(1) Items: 33 items

Initial State: $[Goal \rightarrow \bullet List, eof]$

Desired Final State: $[Goal \rightarrow List \bullet, eof]$

- o The **canonical collection**

$$CC = \{CC_0, CC_1, CC_2, \dots, CC_n\}$$

denotes the set of **valid states** of a **LR(1)** parser.

- Each $CC_i \in CC$ ($0 \leq i \leq n$) is a set of **LR(1) items**.

- $CC \subseteq \mathbb{P}(\text{LR(1) items})$ $|CC|?$ $[|CC| \leq 2^{|\text{LR(1) items}|}]$

- o To model a **LR(1)** parser, we use techniques similar to how we construct a DFA from an NFA (subset construction and ϵ -closure).
- o **Analogies.**
 - ✓ **LR(1) items** \approx states of source *NFA*
 - ✓ $CC \approx$ states of target *DFA*

Constructing \mathcal{CC} : The *closure* Procedure (1)

```
1  ALGORITHM: closure
2  INPUT: CFG  $G = (V, \Sigma, R, S)$ , a set  $s$  of LR(1) items
3  OUTPUT: a set of LR(1) items
4  PROCEDURE:
5   $lastS := \emptyset$ 
6  while ( $lastS \neq s$ ):
7   $lastS := s$ 
8  for  $[A \rightarrow \dots \bullet C \delta, a] \in s$ :
9  for  $C \rightarrow \gamma \in R$ :
10 for  $b \in FIRST(\delta a)$ :
11  $s := s \cup \{ [C \rightarrow \bullet \gamma, b] \}$ 
12 return  $s$ 
```

- **Line 8:** $[A \rightarrow \dots \bullet C \delta, a] \in s$ indicates that the parser's next task is to match $C \delta$ with a lookahead symbol a .
- **Line 9:** Given: matching γ can reduce to C
- **Line 10:** Given: $b \in FIRST(\delta a)$ is a valid lookahead symbol after reducing γ to C
- **Line 11:** Add a new item $[C \rightarrow \bullet \gamma, b]$ into s .
- **Line 6:** Termination is guaranteed.
∴ Each iteration adds ≥ 1 item to s (otherwise $lastS \neq s$ is *false*).

Constructing CC : The *closure* Procedure (2.1)

1	$Goal \rightarrow List$
2	$List \rightarrow List\ Pair$
3	$Pair$
4	$Pair \rightarrow (\underline{Pair} \underline{) }$
5	$(\underline{) }$

Initial State: $[Goal \rightarrow \bullet List, eof]$

Calculate $cc_0 = closure([Goal \rightarrow \bullet List, eof])$.

Constructing \mathcal{CC} : The *goto* Procedure (1)

```
1  ALGORITHM: goto
2  INPUT: a set  $S$  of LR(1) items, a symbol  $x$ 
3  OUTPUT: a set of LR(1) items
4  PROCEDURE:
5   $moved := \emptyset$ 
6  for  $item \in S$ :
7    if  $item = [\alpha \rightarrow \beta \bullet x\delta, a]$  then
8       $moved := moved \cup \{ [\alpha \rightarrow \beta x \bullet \delta, a] \}$ 
9    end
10 return  $closure(moved)$ 
```

Line 7: Given: item $[\alpha \rightarrow \beta \bullet x\delta, a]$ (where x is the next to match)

Line 8: Add $[\alpha \rightarrow \beta x \bullet \delta, a]$ (indicating x is matched) to $moved$

Line 10: Calculate and return $closure(moved)$ as the “next state” from s with a “transition” x .

Constructing CC : The *goto* Procedure (2)

1	$Goal \rightarrow List$
2	$List \rightarrow List Pair$
3	$Pair$
4	$Pair \rightarrow (Pair)$
5	$()$

$$cc_0 = \left\{ \begin{array}{lll} [Goal \rightarrow \bullet List, eof] & [List \rightarrow \bullet List Pair, eof] & [List \rightarrow \bullet List Pair, (] \\ [List \rightarrow \bullet Pair, eof] & [List \rightarrow \bullet Pair, (] & [Pair \rightarrow \bullet (Pair), eof] \\ [Pair \rightarrow \bullet (Pair), (] & [Pair \rightarrow \bullet (), eof] & [Pair \rightarrow \bullet (), (] \end{array} \right\}$$

Calculate $goto(cc_0, ()$.

["next state" from cc_0 taking $()$

Constructing CC : The Algorithm (1)

```
1  ALGORITHM: BuildCC
2  INPUT: a grammar  $G = (V, \Sigma, R, S)$ , goal production  $S \rightarrow S'$ 
3  OUTPUT:
4    (1) a set  $CC = \{cc_0, cc_1, \dots, cc_n\}$  where  $cc_i \subseteq G$ 's LR(1) items
5    (2) a transition function
6  PROCEDURE:
7     $cc_0 := \text{closure}(\{[S' \rightarrow \bullet S, \text{eof}]\})$ 
8     $CC := \{cc_0\}$ 
9     $processed := \{cc_0\}$ 
10    $lastCC := \emptyset$ 
11   while ( $lastCC \neq CC$ ):
12      $lastCC := CC$ 
13     for  $cc_j$  s.t.  $cc_j \in CC \wedge cc_j \notin processed$ :
14        $processed := processed \cup \{cc_j\}$ 
15       for  $x$  s.t.  $[\dots \rightarrow \dots \bullet x \dots] \in cc_j$ 
16          $temp := \text{goto}(cc_j, x)$ 
17         if  $temp \notin CC$  then
18            $CC := CC \cup \{temp\}$ 
19         end
20      $\delta := \delta \cup (cc_j, x, temp)$ 
```

Constructing \mathcal{CC} : The Algorithm (2.1)

1	$Goal \rightarrow List$
2	$List \rightarrow List\ Pair$
3	$Pair$
4	$Pair \rightarrow (\underline{Pair} \underline{)}$
5	$(\underline{)}$

- Calculate $\mathcal{CC} = \{CC_0, CC_1, \dots, CC_{11}\}$
- Calculate the transition function $\delta : \mathcal{CC} \times \Sigma \rightarrow \mathcal{CC}$

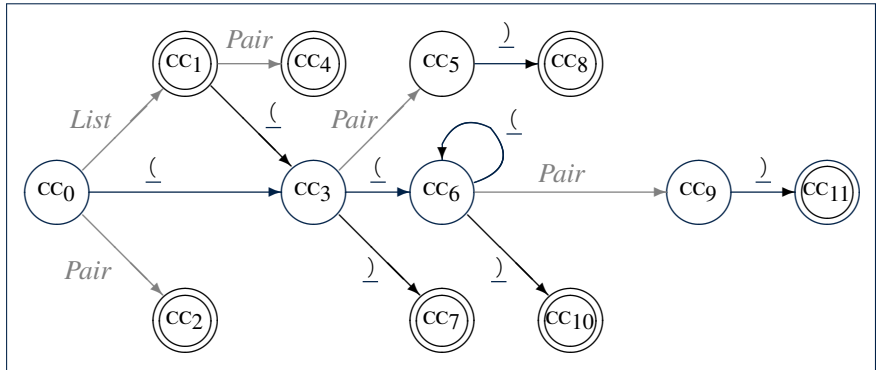
Constructing CC : The Algorithm (2.2)

Resulting transition table:

Iteration	Item	Goal	List	Pair	()	eof
0	CC_0	\emptyset	CC_1	CC_2	CC_3	\emptyset	\emptyset
1	CC_1	\emptyset	\emptyset	CC_4	CC_3	\emptyset	\emptyset
	CC_2	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
	CC_3	\emptyset	\emptyset	CC_5	CC_6	CC_7	\emptyset
2	CC_4	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
	CC_5	\emptyset	\emptyset	\emptyset	\emptyset	CC_8	\emptyset
	CC_6	\emptyset	\emptyset	CC_9	CC_6	CC_{10}	\emptyset
	CC_7	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
3	CC_8	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
	CC_9	\emptyset	\emptyset	\emptyset	\emptyset	CC_{11}	\emptyset
	CC_{10}	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
4	CC_{11}	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset

Constructing \mathcal{CC} : The Algorithm (2.3)

Resulting DFA for the parser:



Constructing \mathcal{CC} : The Algorithm (2.4.1)

Resulting canonical collection \mathcal{CC} :

$$CC_0 = \left\{ \begin{array}{lll} [Goal \rightarrow \bullet List, eof] & [List \rightarrow \bullet List Pair, eof] & [List \rightarrow \bullet List Pair, _] \\ [List \rightarrow \bullet Pair, eof] & [List \rightarrow \bullet Pair, _] & [Pair \rightarrow \bullet _ Pair _, eof] \\ [Pair \rightarrow \bullet _ Pair _, _] & [Pair \rightarrow \bullet _ _, eof] & [Pair \rightarrow \bullet _ _, _] \end{array} \right\}$$

$$CC_1 = \left\{ \begin{array}{lll} [Goal \rightarrow List \bullet, eof] & [List \rightarrow List \bullet Pair, eof] & [List \rightarrow List \bullet Pair, _] \\ [Pair \rightarrow \bullet _ Pair _, eof] & [Pair \rightarrow \bullet _ Pair _, _] & [Pair \rightarrow \bullet _ _, eof] \\ & [Pair \rightarrow \bullet _ _, _] & \end{array} \right\}$$

$$CC_2 = \left\{ [List \rightarrow Pair \bullet, eof] \quad [List \rightarrow Pair \bullet, _] \right\}$$

$$CC_3 = \left\{ \begin{array}{lll} [Pair \rightarrow \bullet _ Pair _, _] & [Pair \rightarrow _ \bullet Pair _, eof] & [Pair \rightarrow _ \bullet Pair _, _] \\ [Pair \rightarrow \bullet _ _, _] & [Pair \rightarrow _ \bullet _, eof] & [Pair \rightarrow _ \bullet _, _] \end{array} \right\}$$

$$CC_4 = \left\{ [List \rightarrow List Pair \bullet, eof] \quad [List \rightarrow List Pair \bullet, _] \right\}$$

$$CC_5 = \left\{ [Pair \rightarrow _ _ Pair \bullet _, eof] \quad [Pair \rightarrow _ _ Pair \bullet _, _] \right\}$$

$$CC_6 = \left\{ \begin{array}{ll} [Pair \rightarrow \bullet _ _ Pair _, _] & [Pair \rightarrow _ \bullet _ Pair _, _] \\ [Pair \rightarrow \bullet _ _, _] & [Pair \rightarrow _ \bullet _, _] \end{array} \right\}$$

$$CC_7 = \left\{ [Pair \rightarrow _ _ \bullet, eof] \quad [Pair \rightarrow _ _ \bullet, _] \right\}$$

$$CC_8 = \left\{ [Pair \rightarrow _ _ Pair _ \bullet, eof] \quad [Pair \rightarrow _ _ Pair _ \bullet, _] \right\}$$

$$CC_9 = \left\{ [Pair \rightarrow _ _ Pair \bullet _, _] \right\}$$

$$CC_{10} = \left\{ [Pair \rightarrow _ _ \bullet, _] \right\}$$

$$CC_{11} = \left\{ [Pair \rightarrow _ _ Pair _ \bullet, _] \right\}$$

Constructing Action and Goto Tables (1)

```
1  ALGORITHM: BuildActionGotoTables
2  INPUT:
3    (1) a grammar  $G = (V, \Sigma, R, S)$ 
4    (2) goal production  $S \rightarrow S'$ 
5    (3) a canonical collection  $CC = \{cc_0, cc_1, \dots, cc_n\}$ 
6    (4) a transition function  $\delta: CC \times \Sigma \rightarrow CC$ 
7  OUTPUT: Action Table & Goto Table
8  PROCEDURE:
9    for  $cc_j \in CC$ :
10   for  $item \in cc_j$ :
11     if  $item = [A \rightarrow \beta \bullet x\gamma, a] \setminus pause \wedge \delta(cc_j, x) = cc_j$  then
12       Action[ $j, x$ ] := shift  $j$ 
13     elseif  $item = [A \rightarrow \beta \bullet, a]$  then
14       Action[ $j, a$ ] := reduce  $A \rightarrow \beta$ 
15     elseif  $item = [S \rightarrow S' \bullet, eof]$  then
16       Action[ $j, eof$ ] := accept
17     end
18   for  $v \in V$ :
19     if  $\delta(cc_j, v) = cc_j$  then
20       Goto[ $j, v$ ] =  $j$ 
21     end
```

- **L12, 13:** Next valid step in discovering A is to match terminal symbol x .
- **L14, 15:** Having recognized β , if current word matches lookahead a , reduce β to A .
- **L16, 17:** Accept if input exhausted and what's recognized reducible to start var. S .
- **L20, 21:** Record consequence of a reduction to non-terminal v from state i

Constructing *Action* and *Goto* Tables (2)

Resulting **Action** and **Goto** tables:

State	<i>Action</i> Table			<i>Goto</i> Table	
	eof	()	<i>List</i>	<i>Pair</i>
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 6	s 7		5
4	r 2	r 2			
5			s 8		
6		s 6	s 10		9
7	r 5	r 5			
8	r 4	r 4			
9			s 11		
10			r 5		
11			r 4		

BUP: Discovering Ambiguity (1)

1	<i>Goal</i>	→	<i>Stmt</i>
2	<i>Stmt</i>	→	if expr then <i>Stmt</i>
3			if expr then <i>Stmt</i> else <i>Stmt</i>
4			assign

- Calculate $\mathcal{CC} = \{cc_0, cc_1, \dots, \}$
- Calculate the transition function $\delta : \mathcal{CC} \times \Sigma \rightarrow \mathcal{CC}$

BUP: Discovering Ambiguity (2.1)

Resulting transition table:

	Item	Goal	Stmt	if	expr	then	else	assign	eof
0	CC ₀	∅	CC ₁	CC ₂	∅	∅	∅	CC ₃	∅
1	CC ₁	∅	∅	∅	∅	∅	∅	∅	∅
	CC ₂	∅	∅	∅	CC ₄	∅	∅	∅	∅
	CC ₃	∅	∅	∅	∅	∅	∅	∅	∅
2	CC ₄	∅	∅	∅	∅	CC ₅	∅	∅	∅
3	CC ₅	∅	CC ₆	CC ₇	∅	∅	∅	CC ₈	∅
4	CC ₆	∅	∅	∅	∅	∅	CC ₉	∅	∅
	CC ₇	∅	∅	∅	CC ₁₀	∅	∅	∅	∅
	CC ₈	∅	∅	∅	∅	∅	∅	∅	∅
5	CC ₉	∅	CC ₁₁	CC ₂	∅	∅	∅	CC ₃	∅
	CC ₁₀	∅	∅	∅	∅	CC ₁₂	∅	∅	∅
6	CC ₁₁	∅	∅	∅	∅	∅	∅	∅	∅
	CC ₁₂	∅	CC ₁₃	CC ₇	∅	∅	∅	CC ₈	∅
7	CC ₁₃	∅	∅	∅	∅	∅	CC ₁₄	∅	∅
8	CC ₁₄	∅	CC ₁₅	CC ₇	∅	∅	∅	CC ₈	∅
9	CC ₁₅	∅	∅	∅	∅	∅	∅	∅	∅

BUP: Discovering Ambiguity (2.2.1)

Resulting canonical collection CC :

$$CC_0 = \left\{ \begin{array}{ll} [Goal \rightarrow \bullet Stmt, eof] & [Stmt \rightarrow \bullet \text{if expr then } Stmt, eof] \\ [Stmt \rightarrow \bullet \text{assign}, eof] & [Stmt \rightarrow \bullet \text{if expr then } Stmt \text{ else } Stmt, eof] \end{array} \right\}$$

$$CC_2 = \left\{ \begin{array}{l} [Stmt \rightarrow \text{if } \bullet \text{ expr then } Stmt, eof], \\ [Stmt \rightarrow \text{if } \bullet \text{ expr then } Stmt \text{ else } Stmt, eof] \end{array} \right\}$$

$$CC_4 = \left\{ \begin{array}{l} [Stmt \rightarrow \text{if expr } \bullet \text{ then } Stmt, eof], \\ [Stmt \rightarrow \text{if expr } \bullet \text{ then } Stmt \text{ else } Stmt, eof] \end{array} \right\}$$

$$CC_6 = \left\{ \begin{array}{l} [Stmt \rightarrow \text{if expr then } Stmt \bullet, eof], \\ [Stmt \rightarrow \text{if expr then } Stmt \bullet \text{ else } Stmt, eof] \end{array} \right\}$$

$$CC_1 = \left\{ [Goal \rightarrow Stmt \bullet, eof] \right\}$$

$$CC_3 = \left\{ [Stmt \rightarrow \text{assign } \bullet, eof] \right\}$$

$$CC_5 = \left\{ \begin{array}{l} [Stmt \rightarrow \text{if expr then } \bullet Stmt, eof], \\ [Stmt \rightarrow \text{if expr then } \bullet Stmt \text{ else } Stmt, eof], \\ [Stmt \rightarrow \bullet \text{if expr then } Stmt, \{eof, else\}], \\ [Stmt \rightarrow \bullet \text{assign}, \{eof, else\}], \\ [Stmt \rightarrow \bullet \text{if expr then } Stmt \text{ else } Stmt, \{eof, else\}] \end{array} \right\}$$

$$CC_7 = \left\{ \begin{array}{l} [Stmt \rightarrow \text{if } \bullet \text{ expr then } Stmt, \{eof, else\}], \\ [Stmt \rightarrow \text{if } \bullet \text{ expr then } Stmt \text{ else } Stmt, \{eof, else\}] \end{array} \right\}$$

BUP: Discovering Ambiguity (2.2.2)

Resulting canonical collection CC :

$$CC_8 = \{[Stmt \rightarrow assign \bullet, \{eof, else\}]\}$$

$$CC_{10} = \left\{ \begin{array}{l} [Stmt \rightarrow if \ expr \ \bullet \ then \ Stmt, \{eof, else\}], \\ [Stmt \rightarrow if \ expr \ then \ Stmt \ else \ Stmt, \{eof, else\}] \end{array} \right\}$$

$$CC_{12} = \left\{ \begin{array}{l} [Stmt \rightarrow if \ expr \ then \ \bullet \ Stmt, \{eof, else\}], \\ [Stmt \rightarrow if \ expr \ then \ \bullet \ Stmt \ else \ Stmt, \{eof, else\}], \\ [Stmt \rightarrow \bullet \ if \ expr \ then \ Stmt, \{eof, else\}], \\ [Stmt \rightarrow \bullet \ if \ expr \ then \ Stmt \ else \ Stmt, \{eof, else\}], \\ [Stmt \rightarrow \bullet \ assign, \{eof, else\}] \end{array} \right\}$$

$$CC_{14} = \left\{ \begin{array}{l} [Stmt \rightarrow if \ expr \ then \ Stmt \ else \ \bullet \ Stmt, \{eof, else\}], \\ [Stmt \rightarrow \bullet \ if \ expr \ then \ Stmt, \{eof, else\}], \\ [Stmt \rightarrow \bullet \ if \ expr \ then \ Stmt \ else \ Stmt, \{eof, else\}], \\ [Stmt \rightarrow \bullet \ assign, \{eof, else\}] \end{array} \right\}$$

$$CC_9 = \left\{ \begin{array}{l} [Stmt \rightarrow if \ expr \ then \ Stmt \ else \ \bullet \ Stmt, eof], \\ [Stmt \rightarrow \bullet \ if \ expr \ then \ Stmt, eof], \\ [Stmt \rightarrow \bullet \ if \ expr \ then \ Stmt \ else \ Stmt, eof], \\ [Stmt \rightarrow \bullet \ assign, eof] \end{array} \right\}$$

$$CC_{11} = \{[Stmt \rightarrow if \ expr \ then \ Stmt \ else \ Stmt \ \bullet, eof]\}$$

$$CC_{13} = \left\{ \begin{array}{l} [Stmt \rightarrow if \ expr \ then \ Stmt \ \bullet \ ., \{eof, else\}], \\ [Stmt \rightarrow if \ expr \ then \ Stmt \ \bullet \ else \ Stmt, \{eof, else\}] \end{array} \right\}$$

BUP: Discovering Ambiguity (3)

- Consider cc_{13}

$$cc_{13} = \left\{ \begin{array}{l} [Stmt \rightarrow \text{if expr then Stmt} \bullet, \{\text{eof, else}\}], \\ [Stmt \rightarrow \text{if expr then Stmt} \bullet \text{ else Stmt}, \{\text{eof, else}\}] \end{array} \right\}$$

Q. What does it mean if the current word to consume is `else`?

A. We can either **shift** (then expecting to match another *Stmt*) or reduce to a *Stmt*.

A single *Action* table entry cannot hold these two alternatives.

This is known as the **shift-reduce conflict**.

- Consider another scenario, say:

$[A \rightarrow \gamma\delta\bullet, a]$

$[B \rightarrow \gamma\delta\bullet, a]$

Q. What does it mean if the current word to consume is `a`?

A. We can either **reduce** to *A* or **reduce** to *B*.

A single *Action* table entry cannot hold these two alternatives.

This is known as the **reduce-reduce conflict**.

Index (1)

Parser in Context

Context-Free Languages: Introduction

CFG: Example (1.1)

CFG: Example (1.2)

CFG: Example (1.2)

CFG: Example (2)

CFG: Example (3)

CFG: Example (4)

CFG: Example (5.1) Version 1

CFG: Example (5.2) Version 1

CFG: Example (5.3) Version 1

Index (2)

CFG: Example (5.4) Version 1

CFG: Example (5.5) Version 2

CFG: Example (5.6) Version 2

CFG: Example (5.7) Version 2

CFG: Formal Definition (1)

CFG: Formal Definition (2): Example

CFG: Formal Definition (3): Example

Regular Expressions to CFG's

DFA to CFG's

CFG: Leftmost Derivations (1)

CFG: Rightmost Derivations (1)

Index (3)

CFG: Leftmost Derivations (2)

CFG: Rightmost Derivations (2)

CFG: Parse Trees vs. Derivations (1)

CFG: Parse Trees vs. Derivations (2)

CFG: Ambiguity: Definition

CFG: Ambiguity: Exercise (1)

CFG: Ambiguity: Exercise (2.1)

CFG: Ambiguity: Exercise (2.2)

CFG: Ambiguity: Exercise (2.3)

Discovering Derivations

TDP: Discovering Leftmost Derivation

Index (4)

TDP: Exercise (1)

TDP: Exercise (2)

Left-Recursions (LF): Direct vs. Indirect

TDP: (Preventively) Eliminating LR

CFG: Eliminating ϵ -Productions (1)

CFG: Eliminating ϵ -Productions (2)

Backtrack-Free Parsing (1)

The first Set: Definition

The first Set: Examples

Computing the first Set

Computing the first Set: Extension

Index (5)

Extended first Set: Examples

Is the first Set Sufficient?

The follow Set: Examples

Computing the follow Set

Backtrack-Free Grammar

TDP: Lookahead with One Symbol

Backtrack-Free Grammar: Exercise

Backtrack-Free Grammar: Left-Factoring

Left-Factoring: Exercise

TDP: Terminating and Backtrack-Free

Backtrack-Free Parsing (2.1)

Index (6)

Backtrack-Free Parsing (2.2)

LL(1) Parser: Exercise

BUP: Discovering Rightmost Derivation

BUP: Discovering Rightmost Derivation (1)

BUP: Discovering Rightmost Derivation (2)

BUP: Example Tracing (1)

BUP: Example Tracing (2.1)

BUP: Example Tracing (2.2)

BUP: Example Tracing (2.3)

LR(1) Items: Definition

LR(1) Items: Scenarios

Index (7)

LR(1) Items: Example (1.1)

LR(1) Items: Example (1.2)

LR(1) Items: Example (1.3)

LR(1) Items: Example (2)

Canonical Collection (CC) vs. LR(1) items

Constructing CC : The *closure* Procedure (1)

Constructing CC : The *closure* Procedure (2.1)

Constructing CC : The *goto* Procedure (1)

Constructing CC : The *goto* Procedure (2)

Constructing CC : The Algorithm (1)

Constructing CC : The Algorithm (2.1)

Index (8)

Constructing \mathcal{CC} : The Algorithm (2.2)

Constructing \mathcal{CC} : The Algorithm (2.3)

Constructing \mathcal{CC} : The Algorithm (2.4)

Constructing *Action* and *Goto* Tables (1)

Constructing *Action* and *Goto* Tables (2)

BUP: Discovering Ambiguity (1)

BUP: Discovering Ambiguity (2.1)

BUP: Discovering Ambiguity (2.2.1)

BUP: Discovering Ambiguity (2.2.2)

BUP: Discovering Ambiguity (3)