

Parsing

Compiler Design

CSE 504

- ① Grammars
- ② Recursive-Descent Parsing
- ③ Top-Down Predictive Parsing
- ④ Universal CFG Parsing

Last modified: Fri Feb 26 2016 at 11:53:22 EST
Version: 1.6 18:44:23 2015/03/06
Compiled at 11:55 on 2016/02/26

Compiler Design

Parsing

CSE 504

1 / 46

Grammars

Parsing

A.k.a. **Syntax Analysis**

- Recognize *sentences* in a language.
- Discover the structure of a document/program.
- Construct (implicitly or explicitly) a tree (called as a *parse tree*) to represent the structure.
- The parse tree is used to guide translation.

Grammars

The syntactic structure of a language is defined using *grammars*.

- Grammars (like regular expressions) specify a set of strings over an alphabet.
- Efficient *recognizers* (automata) can be constructed to determine whether a string is in the language.
- Language hierarchy:
 - Finite Languages (FL)
Enumeration
 - Regular Languages (RL $\supset$ FL)
Regular Expressions
 - Context-Free Languages (CFL $\supset$ RL)
Context-Free Grammars

Regular Languages

Languages represented by regular expressions $\equiv$ Languages recognized by finite automata

Examples:

- ✓ $\{a, b, c\}$
- ✓ $\{\epsilon, a, b, aa, ab, ba, bb, \dots\}$
- ✓ $\{(ab)^n \mid n \geq 0\}$
- ✗ $\{a^n b^n \mid n \geq 0\}$

Context-Free Grammars

- **Terminal Symbols:** Tokens
- **Nonterminal Symbols:** set of strings made up of tokens
- **Productions:** Rules for constructing the set of strings associated with nonterminal symbols.

Example: $Stmt \longrightarrow \text{while } Expr \text{ do } Stmt$

- **Start symbol:** a nonterminal symbol that represents the set of all strings in the language.

Grammars

Grammars

Notation where recursion is explicit.

Examples:

- $\{\epsilon, a, b, aa, ab, ba, bb, \dots\} = \mathcal{L}((a \mid b)^*)$:

S	$\longrightarrow$	ϵ	Notational shorthand:
S	$\longrightarrow$	ES	
E	$\longrightarrow$	a	
E	$\longrightarrow$	b	

S	$\longrightarrow$	$\epsilon \mid ES$
E	$\longrightarrow$	$a \mid b$

- $\{a^n b^n \mid n \geq 0\}$:

$$\begin{array}{lcl} S & \longrightarrow & \epsilon \\ S & \longrightarrow & aSb \end{array}$$

- $\{a^n b^n c^n \mid n \geq 0\}$: Not expressible in CFG .

The First Useful Example

$$\begin{aligned}
 E &\longrightarrow E + E \\
 E &\longrightarrow E - E \\
 E &\longrightarrow E * E \\
 E &\longrightarrow E / E \\
 E &\longrightarrow (E) \\
 E &\longrightarrow \text{id}
 \end{aligned}$$

$$\mathcal{L}(E) = \{\text{id}, \text{id} + \text{id}, \text{id} - \text{id}, \dots, \text{id} + (\text{id} * \text{id}) - \text{id}, \dots\}$$

Context-Free Grammars: Notations

Production: rule with *nonterminal* symbol on left hand side, and a (possibly empty) sequence of terminal or nonterminal symbols on the right hand side.

Notations:

- **Terminals:** lower case letters, digits, punctuation
- **Nonterminals:** Upper case letters
- **Arbitrary Terminals/Nonterminals:** X, Y, Z
- **Strings of Terminals:** u, v, w
- **Strings of Terminals/Nonterminals:** α, β, γ
- **Start Symbol:** S

Derivations

Grammar:

$$E \rightarrow E + E$$

$$E \rightarrow \text{id}$$

E derives $\text{id} + \text{id}$:

$$E \Rightarrow E + E$$

$$\Rightarrow E + \text{id}$$

$$\Rightarrow \text{id} + \text{id}$$

- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ iff $A \rightarrow \gamma$ is a production in the grammar.
- $\alpha \xRightarrow{*} \beta$ if α derives β in zero or more steps.
Example: $E \xRightarrow{*} \text{id} + \text{id}$
- **Sentence:** A sequence of terminal symbols w such that $S \xRightarrow{+} w$ (where S is the start symbol)
- **Sentential Form:** A sequence of terminal/nonterminal symbols α such that $S \xRightarrow{*} \alpha$

Derivations

Grammar:

$$E \rightarrow E + E$$

$$E \rightarrow \text{id}$$

- **Leftmost derivation:** Leftmost nonterminal is replaced first:

$$E \Rightarrow \underline{E} + E$$

$$\Rightarrow \text{id} + \underline{E}$$

$$\Rightarrow \text{id} + \text{id}$$

Written as $E \xRightarrow{*}_{lm} \text{id} + \text{id}$

- **Rightmost derivation:** Rightmost nonterminal is replaced first:

$$E \Rightarrow E + \underline{E}$$

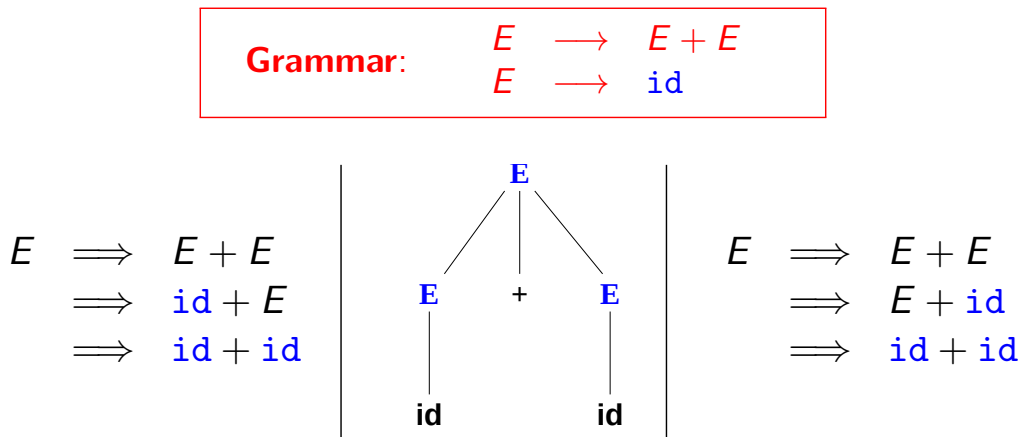
$$\Rightarrow \underline{E} + \text{id}$$

$$\Rightarrow \text{id} + \text{id}$$

Written as $E \xRightarrow{*}_{rm} \text{id} + \text{id}$

Parse Trees

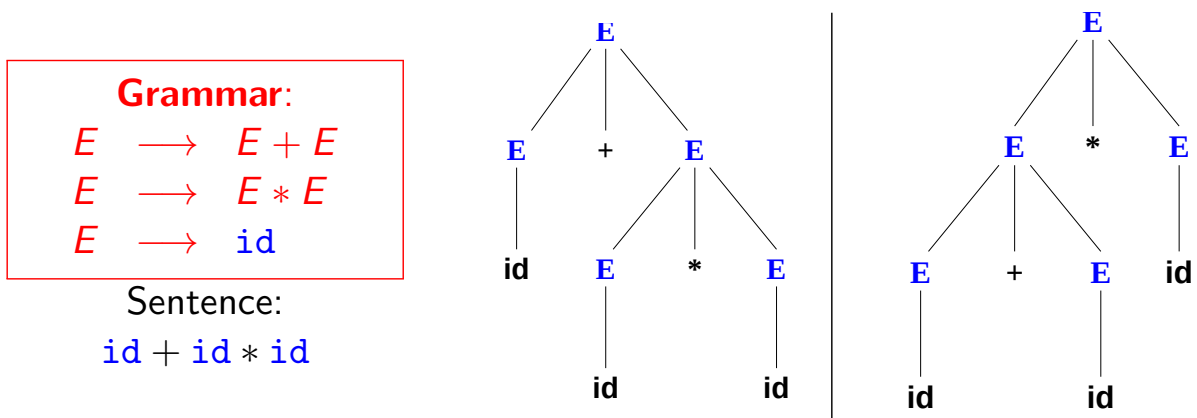
A Parse Tree is a graphical representation of a derivation



A **Parse Tree** succinctly captures the structure of a sentence.

Ambiguity

A Grammar is *ambiguous* if there are multiple parse trees for the same sentence.

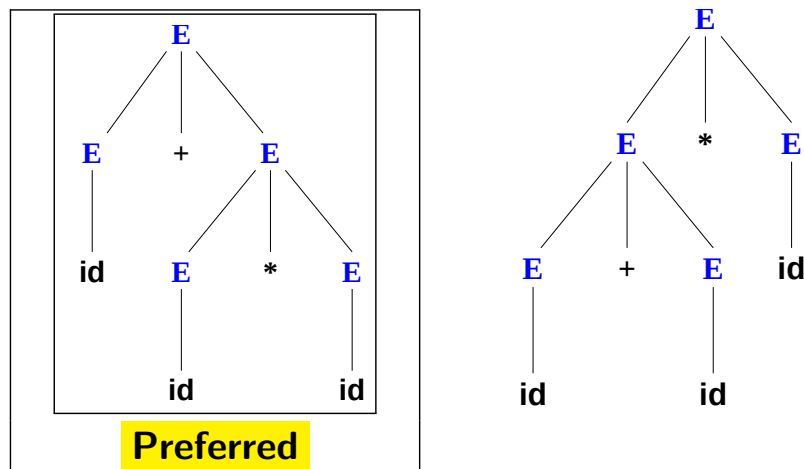


Disambiguation

Express Preference for one parse tree over others.

Example: $\text{id} + \text{id} * \text{id}$

The usual precedence of $*$ over $+$ means:



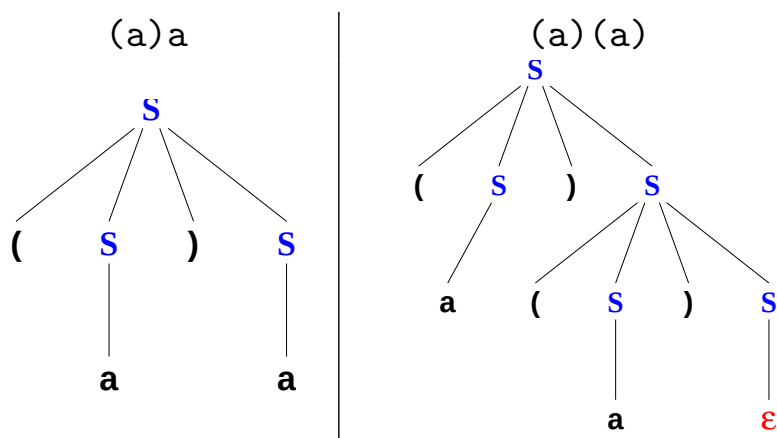
Parsing

Construct a parse tree for a given string.

$$S \rightarrow (S)S$$

$$S \rightarrow a$$

$$S \rightarrow \epsilon$$



A Procedure for Parsing

Grammar: $S \rightarrow a$

```

Algorithm parse_S() {
  switch (input_token) {
    case TOKEN_A:
      consume(TOKEN_A);
      return;
    default:
      /* Parse Error */
  }
}

```

A Procedure for Parsing (Contd.)

Grammar: $S \rightarrow a$
 $S \rightarrow \epsilon$

```

Algorithm parse_S() {
  switch (input_token) {
    case TOKEN_A: /* Production 1 */
      consume(TOKEN_A);
      return;
    case TOKEN_EOF: /* Production 2 */
      return;
    default:
      /* Parse Error */
  }
}

```

A Procedure for Parsing (Contd.)

Grammar:	$S \rightarrow (S)S$
	$S \rightarrow a$
	$S \rightarrow \epsilon$

```

Algorithm parse_S() {
  switch (input_token) {
    case TOKEN_OPEN_PAREN: /* Production 1 */
      consume(TOKEN_OPEN_PAREN);
      parse_S();
      consume(TOKEN_CLOSE_PAREN);
      parse_S();
      return;
  }
  /* Continued on next page */
}

```

A Procedure for Parsing (contd.)

Grammar:	$S \rightarrow (S)S$
	$S \rightarrow a$
	$S \rightarrow \epsilon$

```

    case TOKEN_A: /* Production 2 */
      consume(TOKEN_A);
      return;
    case TOKEN_CLOSE_PAREN:
    case TOKEN_EOF: /* Production 3 */
      return;
    default:
      /* Parse Error */

```

Predictive Parsing: Restrictions

May not be able to choose a unique production

$$\begin{aligned} S &\longrightarrow a B d \\ B &\longrightarrow b \\ B &\longrightarrow bc \end{aligned}$$

In general, we may need a backtracking parser:

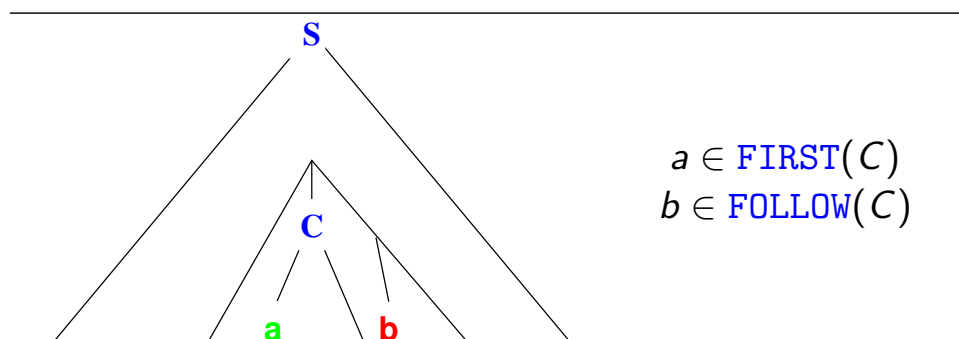
Recursive Descent Parsing

Top-Down Predictive Parsing

FIRST and FOLLOW

Grammar: $S \longrightarrow (S)S \mid a \mid \epsilon$

- **FIRST**(X) = First symbol of any string that can be derived from X
 $\text{FIRST}(S) = \{ (, a, \epsilon \}$.
- **FOLLOW**(A) = First symbol that, in some derivation of a sentence in the language, appears immediately after A .
 $\text{FOLLOW}(S) = \{), \text{EOF} \}$



FIRST

- Let X be a sequence of terminal and non-terminal symbols.
- $\text{FIRST}(X)$ is the *smallest* set such that:
 - If $X \xRightarrow{*} \alpha$ and a is the first terminal symbol in α , then a is in $\text{FIRST}(X)$
 - If $X \xRightarrow{*} \epsilon$ then ϵ is in $\text{FIRST}(X)$

A procedure for computing FIRST is shown later.

FOLLOW

- Let A be a non-terminal symbol, and S be the start symbol of a given grammar.
- $\text{FOLLOW}(A)$ is the *smallest* set such that:
 - If $S \xRightarrow{*} \alpha A \beta$ and a is the first terminal symbol in β , then a is in $\text{FOLLOW}(A)$
 - If $S \xRightarrow{*} \alpha A$, then EOF is in $\text{FOLLOW}(A)$

A procedure for computing FOLLOW is shown later.

FIRST and FOLLOW

Grammar:

$$\begin{array}{ll} S \longrightarrow A S B & A \longrightarrow a \\ S \longrightarrow \epsilon & B \longrightarrow b \end{array}$$

$FIRST(X)$: First terminal in some α such that
 $X \xRightarrow{*} \alpha$.
 $FOLLOW(A)$: First terminal in some β such that
 $S \xRightarrow{*} \alpha A \beta$.

$$\begin{array}{ll} FIRST(S) = \{ a, \epsilon \} & FOLLOW(S) = \{ b, EOF \} \\ FIRST(A) = \{ a \} & FOLLOW(A) = \{ a, b \} \\ FIRST(B) = \{ b \} & FOLLOW(B) = \{ b, EOF \} \end{array}$$

Definition of FIRST

Grammar:

$$\begin{array}{ll} S \longrightarrow A S B & A \longrightarrow a \\ S \longrightarrow \epsilon & B \longrightarrow b \end{array}$$

$FIRST(\alpha)$ is the smallest set such that

$\alpha =$	Property of $FIRST(\alpha)$
a , a terminal	$a \in FIRST(\alpha)$
A , a nonterminal	$A \longrightarrow \epsilon \in G \implies \epsilon \in FIRST(\alpha)$ $A \longrightarrow \beta \in G, \beta \neq \epsilon \implies FIRST(\beta) \subseteq FIRST(\alpha)$
$X_1, X_2, \dots, X_k$, a string of terminals and nonterminals	$FIRST(X_1) - \{\epsilon\} \subseteq FIRST(\alpha)$ $FIRST(X_i) \subseteq FIRST(\alpha)$ if $\forall j < i \quad \epsilon \in FIRST(X_j)$ $\epsilon \in FIRST(\alpha)$ if $\forall j \leq k \quad \epsilon \in FIRST(X_j)$

$FIRST(A) \supseteq FIRST(a) \supseteq \{a\}$ $FIRST(B) \supseteq FIRST(b) \supseteq \{b\}$
 $FIRST(S) \supseteq \{\epsilon\}$, and $FIRST(S) \supseteq FIRST(\{ASB\}) \supseteq FIRST(A) \supseteq \{a\}$

Definition of FOLLOW

Grammar:

$$\begin{array}{ll} S \rightarrow A S B & A \rightarrow a \\ S \rightarrow \epsilon & B \rightarrow b \end{array}$$

$FOLLOW(A)$ is the smallest set such that

A	Property of $FOLLOW(A)$
$= S$, the start symbol	$EOF \in FOLLOW(S)$ Book notation: $\$ \in FOLLOW(S)$
$B \rightarrow \alpha A \beta \in G$	$FIRST(\beta) - \{\epsilon\} \subseteq FOLLOW(A)$
$B \rightarrow \alpha A$, or $B \rightarrow \alpha A \beta, \epsilon \in FIRST(\beta)$	$FOLLOW(B) \subseteq FOLLOW(A)$

$FOLLOW(S) \supseteq \{EOF\}$, and $FOLLOW(S) \supseteq FIRST(B) - \epsilon \supseteq \{b\}$
 $FOLLOW(A) \supseteq FIRST(SB) - \epsilon \supseteq \{a, b\}$
 $FOLLOW(B) \supseteq FOLLOW(S) \supseteq \{b, EOF\}$

Parsing Table

Grammar:

$$\begin{array}{ll} S \rightarrow A S B & A \rightarrow a \\ S \rightarrow \epsilon & B \rightarrow b \end{array}$$

```

Algorithm parse_S() {
  switch (input_token) {
    case TOKEN_A: /* Production 3 */
      parse_A();
      parse_S();
      parse_B();
      return;
    case TOKEN_B:
    case TOKEN_EOF: /* Production 4 */
      return;
  }
}

```

Parsing Table:

NONTERMINAL	INPUT SYMBOL		
	a	b	EOF
S	$S \rightarrow A S B$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$

Table-driven Parsing

Grammar:

$$\begin{array}{ll}
 S \longrightarrow A S B & A \longrightarrow a \\
 S \longrightarrow \epsilon & B \longrightarrow b
 \end{array}$$

Parsing Table:

NONTERMINAL	INPUT SYMBOL		
	a	b	EOF
S	$S \longrightarrow A S B$	$S \longrightarrow \epsilon$	$S \longrightarrow \epsilon$
A	$A \longrightarrow a$		
B		$B \longrightarrow b$	

Nonrecursive Parsing

Instead of recursion,

use an explicit *stack* along with the parsing table.

Data objects:

- **Parsing Table:** $M(A, a)$, a two-dimensional array, dimensions indexed by nonterminal symbols (A) and terminal symbols (a).
- A **Stack** of terminal/nonterminal symbols
- **Input stream** of tokens

The above data structures manipulated using a *table-driven parsing program*.

Table-driven Parsing: a Sketch

stack initialized to hold the start symbol.

```

while (! stack.isEmpty()) {
    X = stack.top();
    if (X is a terminal symbol)
        consume(X);
    else /* X is a nonterminal */
        if ( $M[X, input\_token] = X \longrightarrow Y_1, Y_2, \dots, Y_k$ ) {
            stack.pop();
            for i = k downto 1 do
                stack.push(Yi);
        }
    else
        /* Syntax Error */
}

```

Constructing Parsing Table

Grammar:

$S \longrightarrow A S B$	$A \longrightarrow a$
$S \longrightarrow \epsilon$	$B \longrightarrow b$

$First(S) = \{ a, \epsilon \}$	$Follow(S) = \{ b, EOF \}$
$First(A) = \{ a \}$	$Follow(A) = \{ a, b \}$
$First(B) = \{ b \}$	$Follow(B) = \{ b, EOF \}$

NONTERMINAL	INPUT SYMBOL		
	a	b	EOF
<i>S</i>	$S \longrightarrow A S B$	$S \longrightarrow \epsilon$	$S \longrightarrow \epsilon$
<i>A</i>	$A \longrightarrow a$		
<i>B</i>		$B \longrightarrow b$	

A Procedure to Construct Parsing Tables

$FIRST(X)$: First terminal in some α such that
 $X \xRightarrow{*} \alpha$.
 $FOLLOW(A)$: First terminal in some β such that
 $S \xRightarrow{*} \alpha A \beta$.

```

Algorithm table_construct( $G$ ) {
  for each  $A \rightarrow \alpha \in G$  {
    for each  $a \in FIRST(\alpha)$  such that  $a \neq \epsilon$ 
      add  $A \rightarrow \alpha$  to  $M[A, a]$ ;
    if  $\epsilon \in FIRST(\alpha)$ 
      for each  $b \in FOLLOW(A)$ 
        add  $A \rightarrow \alpha$  to  $M[A, b]$ ;
  }
}
  
```

Parsing Table: Another Example

Grammar:
 $S \rightarrow (S)S$
 $S \rightarrow a$
 $S \rightarrow \epsilon$

$FIRST(S) = \{ '(', a, \epsilon \}$
 $FOLLOW(S) = \{ ')', EOF \}$

INPUT SYMBOL			
a	(	)	EOF
$S \rightarrow a$	$S \rightarrow (S)S$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$

LL(1) Grammar: When the grammar's recursive descent parsing table has no conflicts (i.e. each cell has at most one entry).

Recursive Descent Parsing: Restrictions

Grammar cannot be left-recursive

Example: $E \rightarrow E + E \mid a$

```

Algorithm parse_E() {
  switch (input_token) {
    case TOKEN_A: /* Production 1 */
      parse_E();
      consume(TOKEN_PLUS);
      parse_E();
      return;
    case TOKEN_A: /* Production 2 */
      consume(TOKEN_A);
      return;
  }
}

```

Removing Left Recursion

$$\begin{aligned}
 A &\rightarrow A a \\
 A &\rightarrow b
 \end{aligned}$$

$$\mathcal{L}(A) = \{b, ba, baa, baaa, baaaa, \dots\}$$

$$\begin{aligned}
 A &\rightarrow bA' \\
 A' &\rightarrow aA' \\
 A' &\rightarrow \epsilon
 \end{aligned}$$

Removing Left Recursion: Another Example

$$\begin{aligned} E &\longrightarrow E + E \\ E &\longrightarrow \text{id} \end{aligned}$$

$$\Downarrow$$

$$\begin{aligned} E &\longrightarrow \text{id } E' \\ E' &\longrightarrow + \text{id } E' \\ E' &\longrightarrow \epsilon \end{aligned}$$

Parsing with LL(1) Grammars: Example 1

	a	b	EOF
S	$S \longrightarrow A S B$	$S \longrightarrow \epsilon$	$S \longrightarrow \epsilon$
A	$A \longrightarrow a$		
B		$B \longrightarrow b$	

Stack	Input Stream	Rule	Derivation
\$S	a a b b\$	$S \longrightarrow A S B$	$S \Rightarrow A S B$
\$B S A	a a b b\$	$A \longrightarrow a$	$\Rightarrow a S B$
\$B S a	a a b b\$		
\$B S	a b b\$	$S \longrightarrow A S B$	$\Rightarrow a A S B B$
\$B B S A	a b b\$	$A \longrightarrow a$	$\Rightarrow a a S B B$
\$B B S a	a b b\$		
\$B B S	b b\$	$S \longrightarrow \epsilon$	$\Rightarrow a a B B$
\$B B	b b\$	$B \longrightarrow b$	$\Rightarrow a a b B$
\$B b	b b\$		
\$B	b\$	$B \longrightarrow b$	$\Rightarrow a a b b$
\$b	b\$		
\$	\$		

Parsing with LL(1) Grammars: Example 2

NONTERMINAL	INPUT SYMBOL		
	id	+	EOF
E	$E \rightarrow \text{id } E'$		
E'		$E' \rightarrow + E E'$	$E' \rightarrow \epsilon$

Stack	Input Stream	Rule	Derivation
$\$E$	id + id\$	$E \rightarrow \text{id } E'$	$E \Rightarrow \text{id } E'$
$\$E' \text{ id}$	id + id\$		
$\$E'$	+ id\$	$E' \rightarrow + \text{id } E'$	$\Rightarrow \text{id} + \text{id } E'$
$\$E' \text{ id} +$	+ id\$		
$\$E' \text{ id}$	id\$		
$\$E'$	\$	$E' \rightarrow \epsilon$	$\Rightarrow \text{id} + \text{id}$
\$	\$		

LL(1) Derivations

Left to Right Scan of input

Leftmost Derivation

(1) look ahead 1 token at each step

Alternative characterization of LL(1) Grammars:

Whenever $A \rightarrow \alpha \mid \beta \in G$

① $FIRST(\alpha) \cap FIRST(\beta) = \{ \}$, and

② if $\alpha \xRightarrow{*} \epsilon$ then $FIRST(\beta) \cap FOLLOW(A) = \{ \}$.

Corollary: No Ambiguous Grammar is LL(1).

Other Parsing Algorithms

- LR-, LALR-, SLR, ...
Table-driven “bottom-up” parsers (builds parse trees from leaves to root).
Parsing time is *linear* in the length of the input string.
- The set of $LR(k)$ grammars includes all $LL(k)$ grammars (but some grammars are not $LR(k)$ for any k).
- Operator precedence parsers
- Chart parsers (used in Natural Language Processing)
- Cocke-Kasami-Younger & Earley parsers: can handle arbitrary context-free grammars
(but the parsers may take quadratic or cubic time).

Universal CFG Parsing

Chomsky Normal Form

A context free grammar G is in *Chomsky Normal Form* (CNF) if:

- it is ϵ -free, and
- every production in G is of the form
 - $A \rightarrow BC$, where B and C are non-terminals, or
 - $A \rightarrow a$, where a is a terminal.

A CFG is ϵ -free if it is not possible to derive ϵ from any symbol other than the start symbol of the grammar.

- For every CFG G , there is an equivalent grammar G' that is ϵ -free.
- For every CFG G , there is an equivalent grammar G' that is in Chomsky Normal Form.

Example

Even-Length Palindromes

$$\begin{aligned} S &\rightarrow a S a \\ S &\rightarrow b S b \\ S &\rightarrow \epsilon \end{aligned}$$

Same Lang., ϵ -free

$$\begin{aligned} S &\rightarrow T \\ S &\rightarrow \epsilon \\ T &\rightarrow a T a \\ T &\rightarrow b T b \\ T &\rightarrow a a \\ T &\rightarrow b b \end{aligned}$$

Same Lang., in CNF

$$\begin{aligned} S &\rightarrow A U \\ S &\rightarrow B V \\ S &\rightarrow \epsilon \\ T &\rightarrow A U \\ T &\rightarrow B V \\ U &\rightarrow T A \\ U &\rightarrow a \\ V &\rightarrow T B \\ V &\rightarrow b \\ A &\rightarrow a \\ B &\rightarrow b \end{aligned}$$

Universal CFG Parsing

Consider a grammar G in Chomsky Normal Form.

- Let s represent a sequence of terminals of length n .
- Let $s[i : j]$ be the subsequence made of terminal symbols, from index i to (but not including) index j of s .
 - For instance if $s = abcdef$, then $s[2 : 5] = cde$.
- $A \Rightarrow^* BC \Rightarrow^* s$ is a derivation ...
 - (i.e., s is derivable from A , and the first step in the derivation is the use of production $A \rightarrow BC$)
 - ... then $\exists k$. such that $0 \leq k \leq n$ and $B \Rightarrow^* s[0 : k]$ and $C \Rightarrow^* s[k : n]$.
- In general:

$$A \Rightarrow^* s[i : j] \text{ iff } \begin{cases} \exists A \rightarrow a & s[i : j] = a. \\ \exists A \rightarrow BC, \exists k \text{ s.t. } i < k < j \\ & B \Rightarrow^* s[i : k], \quad C \Rightarrow^* s[k : j]. \end{cases}$$

Cocke-Kasami-Younger Algorithm— An Overview

- Let the sequence of terminals to be parsed be s , of length n .
- Make a two dimensional table $P[n + 1][n + 1]$.
- Each element $P[i][j]$ of this table is a set of **non-terminal** symbols such that

$$A \in P[i][j] \text{ iff } A \xRightarrow{*} s[i : j]$$

- The string s is in the language of the given grammar iff S , the start symbol of the grammar $\in P[0][n]$.

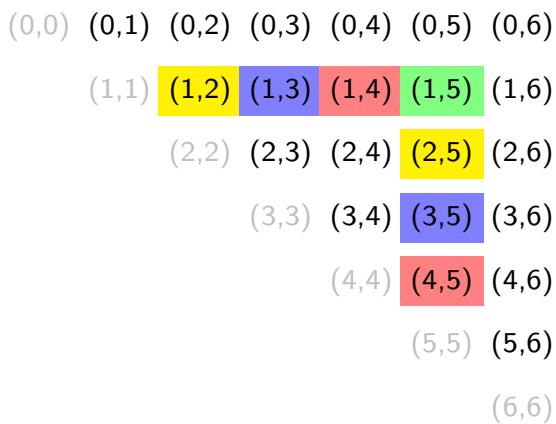
CKY Algorithm— The Details

```

1: procedure CKY( $G, s$ )
2:   for  $i$  from 0 to  $n - 1$  do                                     ▷ Populate superdiagonal
3:     if  $a = s[i : i + 1]$  and  $A \rightarrow a \in G$  then
4:       add  $A$  to  $P[i][i + 1]$ 
5:     end if
6:   end for
7:   for  $\ell$  from 2 to  $n$  do                                         ▷ Do the rest of the matrix
8:     for  $i$  from 0 to  $n - \ell$  do                                     ▷ moving diagonally up
9:        $j = i + \ell$ 
10:      for  $k$  from  $i + 1$  to  $j - 1$  do
11:        if  $B \in P[i][k], C \in P[k][j]$  s.t.  $A \rightarrow BC \in G$  then
12:          add  $A$  to  $P[i][j]$                                        ▷ Note  $k, B, C$  too for parse tree
13:        end if
14:      end for
15:    end for
16:  end for
17:  return  $start(G) \in P[n][n]$ 
18: end procedure

```

Computation in CKY



CKY Algorithm— An Example

Let $s = abbbba$

$S \rightarrow A U$		0	1	2	3	4	5	6
$S \rightarrow B V$	0	{S}	{A,U}					{T,S}
$S \rightarrow \epsilon$	1		{S}	{B,V}	{T,S}	{V}	{T,S}	{U}
$T \rightarrow A U$	2			{S}	{B,V}	{T,S}	{V}	
$T \rightarrow B V$	3				{S}	{B,V}	{T,S}	{U}
$U \rightarrow T A$	4					{S}	{B,V}	
$U \rightarrow a$	5						{S}	{A,U}
$V \rightarrow T B$	6							{S}
$V \rightarrow b$								
$A \rightarrow a$								
$B \rightarrow b$								