Parsing

Compiler Design

CSE 504

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

Last modifled: Fri Mar 06 2015 at 12:23:11 EST
Version: 1.6 18:44:23 2015/03/06
Compiled at 13:45 on 2015/03/06

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 heirarchy:
  - 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:

$\surd$ $\{a, b, c\}$

$\surd$ $\{\epsilon, a, b, aa, ab, ba, bb, \ldots\}$

$\surd$ $\{(ab)^n \mid n \geq 0\}$

$\times$ $\{a^n b^n \mid n \geq 0\}$

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* $\longrightarrow$ `while` *Expr* `do` *Stmt*
- **Start symbol:** a nonterminal symbol that represents the set of all strings in the language.

Grammars

Notation where recursion is explicit.

Examples:

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

  $$\begin{array}{lcl} \mathbf{S} & \longrightarrow & \epsilon \\ \mathbf{S} & \longrightarrow & \mathbf{ES} \\ \mathbf{E} & \longrightarrow & \mathtt{a} \\ \mathbf{E} & \longrightarrow & \mathtt{b} \end{array}$$

  Notational shorthand:

  $$\begin{array}{lcl} \mathbf{S} & \longrightarrow & \epsilon \mid \mathbf{ES} \\ \mathbf{E} & \longrightarrow & \mathtt{a} \mid \mathtt{b} \end{array}$$
- $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$:

  $$\begin{array}{lcl} S & \longrightarrow & \epsilon \\ S & \longrightarrow & \mathtt{a}S\mathtt{b} \end{array}$$
- $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$: Not expressible in CFG.

The First Useful Example

$$
\begin{array}{lcl}
E & \longrightarrow & E + E \\
E & \longrightarrow & E - E \\
E & \longrightarrow & E * E \\
E & \longrightarrow & E \ / \ E \\
E & \longrightarrow & (\ E\ ) \\
E & \longrightarrow & \texttt{id}
\end{array}
$$

$$\mathcal{L}(E) = \{\texttt{id}, \texttt{id} + \texttt{id}, \texttt{id} - \texttt{id}, \ldots, \texttt{id} + (\texttt{id} * \texttt{id}) - \texttt{id}, \ldots\}$$

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:

$$E \longrightarrow E + E$$
$$E \longrightarrow \texttt{id}$$

E **derives** $\texttt{id} + \texttt{id}$:

$$\begin{aligned} E &\Longrightarrow E + E \\ &\Longrightarrow E + \texttt{id} \\ &\Longrightarrow \texttt{id} + \texttt{id} \end{aligned}$$

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

Derivations

Grammar:

$$E \longrightarrow E + E$$
$$E \longrightarrow \texttt{id}$$

- Leftmost derivation: Leftmost nonterminal is replaced first:

$$\begin{aligned} E &\Longrightarrow \underline{E} + E \\ &\Longrightarrow \texttt{id} + \underline{E} \\ &\Longrightarrow \texttt{id} + \texttt{id} \end{aligned}$$

  Written as $E \overset{*}{\Longrightarrow}_{lm} \texttt{id} + \texttt{id}$
- Rightmost derivation: Rightmost nonterminal is replaced first:

$$\begin{aligned} E &\Longrightarrow E + \underline{E} \\ &\Longrightarrow \underline{E} + \texttt{id} \\ &\Longrightarrow \texttt{id} + \texttt{id} \end{aligned}$$

  Written as $E \overset{*}{\Longrightarrow}_{rm} \texttt{id} + \texttt{id}$

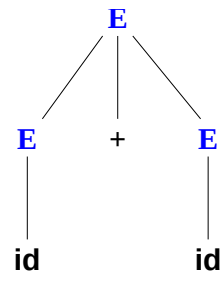
Parse Trees

A Parse Tree is a graphical representation of a derivation

Grammar:

$$E \longrightarrow E + E$$
$$E \longrightarrow \texttt{id}$$

$$\begin{aligned} E &\Longrightarrow E + E \\ &\Longrightarrow \texttt{id} + E \\ &\Longrightarrow \texttt{id} + \texttt{id} \end{aligned}$$



$$\begin{aligned} E &\Longrightarrow E + E \\ &\Longrightarrow E + \texttt{id} \\ &\Longrightarrow \texttt{id} + \texttt{id} \end{aligned}$$

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

Ambiguity

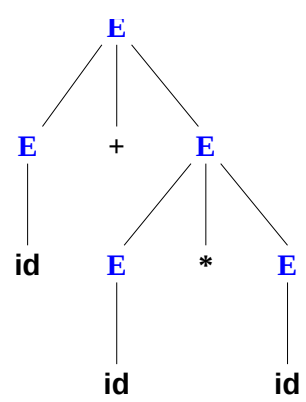
A Grammar is *ambiguous* if there are multiple parse trees for the same sentence.

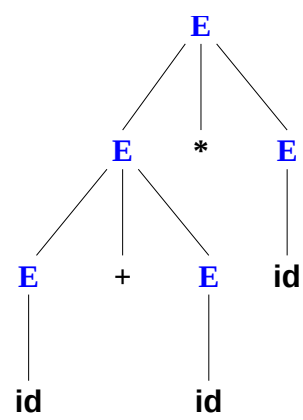
Grammar:

$$E \longrightarrow E + E$$
$$E \longrightarrow E * E$$
$$E \longrightarrow \texttt{id}$$

Sentence:

$\texttt{id} + \texttt{id} * \texttt{id}$



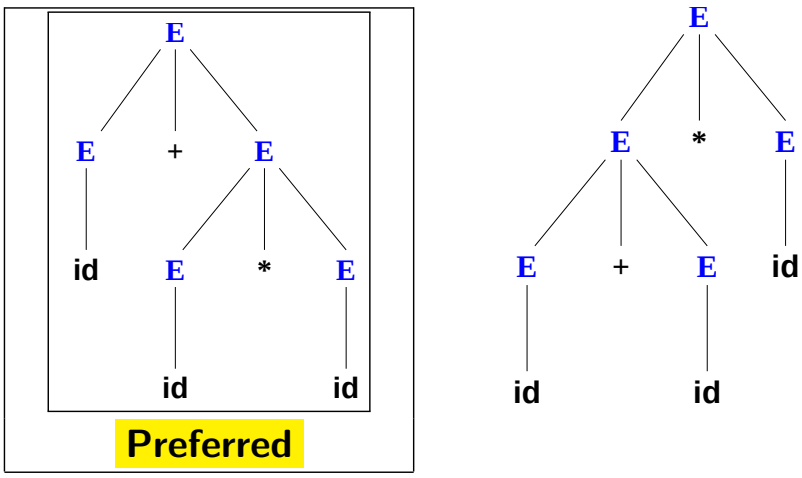


Disambiguition

Express Preference for one parse tree over others.

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

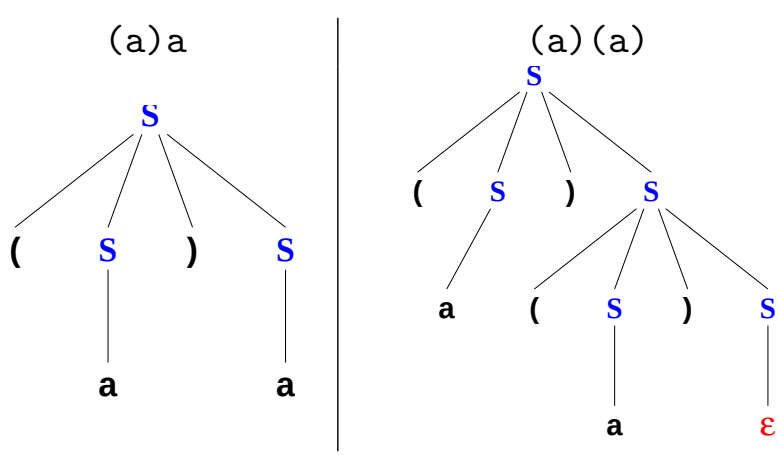
The usual precedence of $*$ over $+$ means:



Parsing

Construct a parse tree for a given string.

$$
\begin{aligned}
S &\longrightarrow (S)S \\
S &\longrightarrow a \\
S &\longrightarrow \epsilon
\end{aligned}
$$



A Procedure for Parsing

Grammar: $S \longrightarrow a$

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


A Procedure for Parsing (Contd.)

Grammar:
$S \longrightarrow a$
$S \longrightarrow \epsilon$

```
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:

$$S \longrightarrow (S)S$$
$$S \longrightarrow a$$
$$S \longrightarrow \epsilon$$

```
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:

$$S \longrightarrow (S)S$$
$$S \longrightarrow a$$
$$S \longrightarrow \epsilon$$

```
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

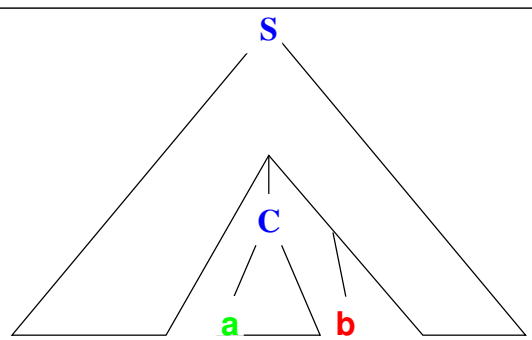
FIRST and FOLLOW

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

- FIRST(X) = First symbol of any string that can be derived from X

  FIRST(S) = {(, a, ϵ}.
- FOLLOW(A) = First symbol that, in some derivation of a sentence in the language, appears immediately after A.

  FOLLOW(S) = {), EOF}



$a \in$ FIRST(C)
$b \in$ FOLLOW(C)

FIRST

- Let X be a sequence of terminal and non-terminal symbols.
- FIRST(X) is the *smallest* set such that:
  - If $X \stackrel{*}{\Longrightarrow} \alpha$ and a is the first terminal symbol in α, then a is in FIRST(X)
  - If $X \stackrel{*}{\Longrightarrow} \epsilon$ then ϵ is in 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.
- FOLLOW(A) is the *smallest* set such that:
  - If $S \stackrel{*}{\Longrightarrow} \alpha A \beta$ and a is the first terminal symbol in β, then a is in FOLLOW(X)
  - If $S \stackrel{*}{\Longrightarrow} \alpha A$, then EOF is in FOLLOW($X$)

A procedure for computing FOLLOW is shown later.

FIRST and FOLLOW

Grammar:

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

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

$$FIRST(S) = \{\ \mathtt{a},\ \epsilon\ \} \qquad FOLLOW(S) = \{\ \mathtt{b},\ \mathrm{EOF}\ \}$$
$$FIRST(A) = \{\ \mathtt{a}\ \} \qquad FOLLOW(A) = \{\ \mathtt{a},\ \mathtt{b}\ \}$$
$$FIRST(B) = \{\ \mathtt{b}\ \} \qquad FOLLOW(B) = \{\ \mathtt{b},\ \mathrm{EOF}\ \}$$

Definition of FIRST

Grammar:

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

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

$\alpha =$	Property of $FIRST(\alpha)$
a, a terminal	$a \in FIRST(\alpha)$
A, a nonterminal	$A \longrightarrow \epsilon \in G \Longrightarrow \epsilon \in FIRST(\alpha)$ $A \longrightarrow \beta \in G,\ \beta \neq \epsilon \Longrightarrow FIRST(\beta) \subseteq FIRST(\alpha)$
$X_1, X_2, \ldots, X_k$, 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:

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

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

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

$FOLLOW(S) \supseteq \{EOF\}$, 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:

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

```
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	$S \longrightarrow A\ S\ B$	$S \longrightarrow \epsilon$	$S \longrightarrow \epsilon$

Table-driven Parsing

Grammar:

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

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 ⟶ Y1, Y2, ..., Yk) {
      stack.pop();
      for i = k downto 1 do
        stack.push(Yi);
    }
    else
      /* Syntax Error */
}
```


Constructing Parsing Table

Grammar:

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

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

Nonterminal	Input Symbol		
	a	b	EOF
S	$S \longrightarrow A\ S\ B$	$S \longrightarrow \epsilon$	$S \longrightarrow \epsilon$
A	$A \longrightarrow a$		
B		$B \longrightarrow b$	

A Procedure to Construct Parsing Tables

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

```
Algorithm table_construct(G) {
  for each A ⟶ α ∈ G {
    for each a ∈ FIRST(α) such that a ≠ ϵ
      add A ⟶ α to M[A, a];
    if ϵ ∈ FIRST(α)
      for each b ∈ FOLLOW(A)
        add A ⟶ α to M[A, b];
}}
```


Parsing Table: Another Example

Grammar:

$$S \longrightarrow (S)S$$
$$S \longrightarrow a$$
$$S \longrightarrow \epsilon$$

FIRST(S) = {'(', a, ϵ}
FOLLOW(S) = {')', EOF}

Input Symbol			
a	(	)	EOF
$S \longrightarrow a$	$S \longrightarrow (S)S$	$S \longrightarrow \epsilon$	$S \longrightarrow \epsilon$

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: $E \longrightarrow E + E \mid a$

```
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

$$A \longrightarrow A\ a$$
$$A \longrightarrow b$$

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

$$A \longrightarrow bA'$$

$$A' \longrightarrow aA'$$
$$A' \longrightarrow \epsilon$$

Removing Left Recursion: Another Example

$$E \longrightarrow E + E$$
$$E \longrightarrow \texttt{id}$$

$$\Downarrow$$

$$E \longrightarrow \texttt{id}\ E'$$
$$E' \longrightarrow +\ \texttt{id}\ E'$$
$$E' \longrightarrow \epsilon$$

Parsing with LL(1) Grammars: Example 1

	a	b	EOF
S	$S \longrightarrow A\ S\ B$	$S \longrightarrow \epsilon$	$S \longrightarrow \epsilon$
A	$A \longrightarrow \texttt{a}$		
B		$B \longrightarrow \texttt{b}$	

Stack	Input Stream	Rule	Derivation
$\$S$	a a b b$	$S \longrightarrow A\ S\ B$	$S \Longrightarrow A\ S\ B$
$\$B\ S\ A$	a a b b$	$A \longrightarrow \texttt{a}$	$\Longrightarrow \texttt{a}\ S\ B$
$\$B\ S\ \texttt{a}$	a a b b$		
$\$B\ S$	a b b$	$S \longrightarrow A\ S\ B$	$\Longrightarrow \texttt{a}\ A\ S\ B\ B$
$\$B\ B\ S\ A$	a b b$	$A \longrightarrow \texttt{a}$	$\Longrightarrow \texttt{a}\ \texttt{a}\ S\ B\ B$
$\$B\ B\ S\ \texttt{a}$	a b b$		
$\$B\ B\ S$	b b$	$S \longrightarrow \epsilon$	$\Longrightarrow \texttt{a}\ \texttt{a}\ B\ B$
$\$B\ B$	b b$	$B \longrightarrow \texttt{b}$	$\Longrightarrow \texttt{a}\ \texttt{a}\ \texttt{b}\ B$
$\$B\ \texttt{b}$	b b$		
$\$B$	b$	$B \longrightarrow \texttt{b}$	$\Longrightarrow \texttt{a}\ \texttt{a}\ \texttt{b}\ \texttt{b}$
$\$\texttt{b}$	b$		
$\$$	$		

Parsing with LL(1) Grammars: Example 2

NONTERMINAL	INPUT SYMBOL		
	id	+	EOF
E	$E \longrightarrow \texttt{id}\ E'$		
E'		$E' \longrightarrow \texttt{+}\ E\ E'$	$E' \longrightarrow \epsilon$

Stack	Input Stream	Rule	Derivation
$E	id + id$	$E \longrightarrow \texttt{id}\ E'$	$E \Longrightarrow \texttt{id}\ E'$
$E' id	id + id$		
$E'	+ id$	$E' \longrightarrow \texttt{+ id}\ E'$	$\Longrightarrow \texttt{id + id}\ E'$
$E' id +	+ id$		
$E' id	id$		
$E'	$	$E' \longrightarrow \epsilon$	$\Longrightarrow$ id+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 $A \longrightarrow \alpha \mid \beta \in G$

1. $FIRST(\alpha) \cap FIRST(\beta) = \{\ \}$, and
2. if $\alpha \overset{*}{\Longrightarrow} \epsilon$ 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).

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

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

Same Lang., ϵ-free

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

Same Lang., in CNF

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

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 $s[2:5] = cde$.
- $A \Rightarrow BC \overset{*}{\Rightarrow} s$ 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 $B \overset{*}{\Rightarrow} s[0:k]$ and $C \overset{*}{\Rightarrow} s[k:n]$.
- In general:

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

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

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

- 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

```
procedure CKY(G, s)
    for i from 0 to n − 1 do                          ▷ Populate superdiagonal
        if a = s[i : i + 1] and A → a ∈ G then
            add A to P[i][i + 1]
        end if
    end for
    for ℓ from 2 to n do                              ▷ Do the rest of the matrix
        for i from 0 to n − ℓ do                      ▷ moving diagonally up
            j = i + ℓ
            for k from i + 1 to j − 1 do
                if B ∈ P[i][k], C ∈ P[k][j] s.t. A → BC ∈ G then
                    add A to P[i][j]                  ▷ Note k, B, C too for parse tree
                end if
            end for
        end for
    end for
    return start(G) ∈ P[n][n]
end procedure
```

Computation in CKY

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)
	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)
		(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
			(3,3)	(3,4)	(3,5)	(3,6)
				(4,4)	(4,5)	(4,6)
					(5,5)	(5,6)
						(6,6)

CKY Algorithm— An Example

Let $s = abbbba$

$S \rightarrow A\ U$
$S \rightarrow B\ V$
$S \rightarrow \epsilon$
$T \rightarrow A\ U$
$T \rightarrow B\ V$
$U \rightarrow T\ A$
$U \rightarrow a$
$V \rightarrow T\ B$
$V \rightarrow b$
$A \rightarrow a$
$B \rightarrow b$

	0	1	2	3	4	5	6
0	{S}	{A,U}					{T,S}
1		{S}	{B,V}	{T,S}	{V}	{T,S}	{U}
2			{S}	{B,V}	{T,S}	{V}	
3				{S}	{B,V}	{T,S}	{U}
4					{S}	{B,V}	
5						{S}	{A,U}
6							{S}