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:

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

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

  Notational shorthand:

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

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

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

- $\{w \mid$ *no. of* $\mathtt{a}$*'s in* $w =$ *no. of* $\mathtt{b}$*'s in* $w\}$:
  Not expressible in CFG.

The First Useful Example

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

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

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:

$$\begin{array}{lcl} E & \longrightarrow & E + E \\ E & \longrightarrow & \texttt{id} \end{array}$$

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

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

- $\alpha A \beta \Longrightarrow \alpha \gamma \beta$ 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 $E \overset{*}{\Longrightarrow}_{rm} \texttt{id} + \texttt{id}$

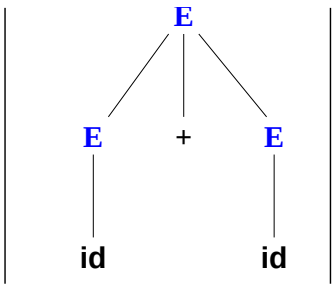
Parse Trees

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

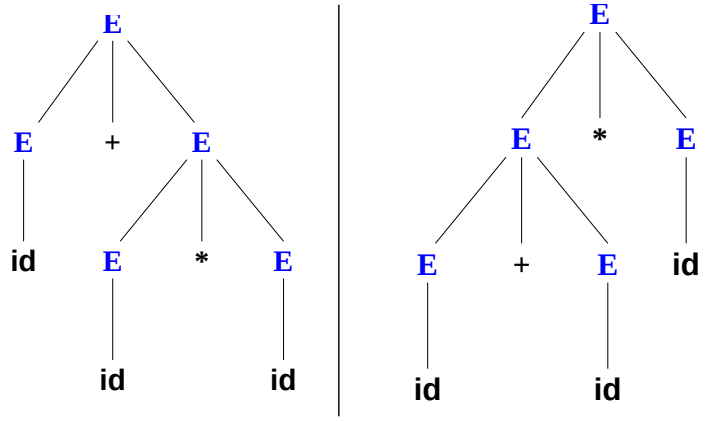
A Grammar is *ambiguous* if there are multiple parse trees for the same sentence.

Grammar:

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

Sentence:

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

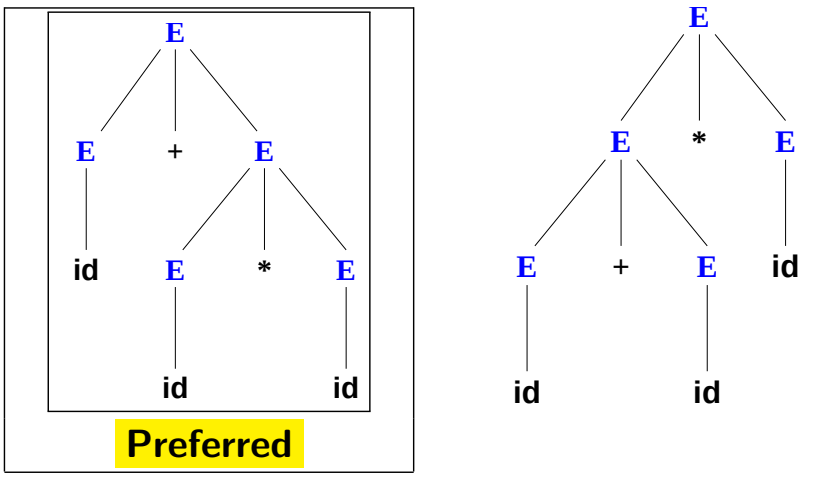


Disambiguition

Express Preference for one parse tree over others.

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

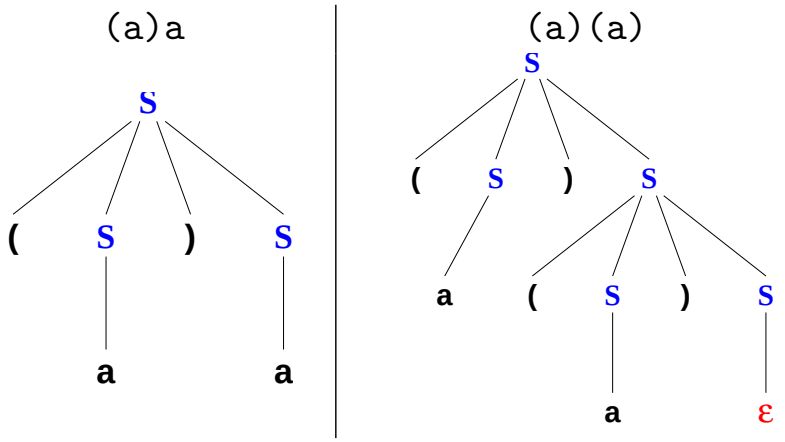
The usual precedence of $*$ over $+$ means:



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

$$S \longrightarrow a\ B\ d$$
$$B \longrightarrow b$$
$$B \longrightarrow bc$$

In general, we may need a backtracking parser:
Recursive Descent Parsing

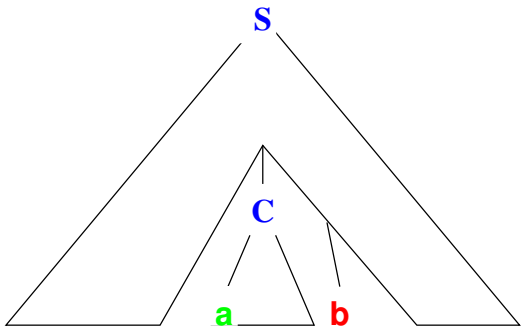
FIRST and FOLLOW

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

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

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

  FOLLOW(S) = {), EOF}



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

FIRST and FOLLOW

Grammar: $A \longrightarrow a$ $\quad S \longrightarrow A\ S\ B$
$B \longrightarrow b$ $\quad S \longrightarrow \epsilon$

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

$FIRST(S)$	=	{ a, ϵ }	$FOLLOW(S)$	=	{ b, EOF }
$FIRST(A)$	=	{ a }	$FOLLOW(A)$	=	{ a, b }
$FIRST(B)$	=	{ b }	$FOLLOW(B)$	=	{ b, EOF }

Definition of FIRST

Grammar:

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

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

$\alpha =$	Property of $FIRST(\alpha)$
a, a terminal	$a \in FIRST(\alpha)$
A, a nonterminal	$A \longrightarrow \epsilon \in G \Longrightarrow \epsilon \in FIRST(\alpha)$ $A \longrightarrow \beta \in G,\ \beta \neq \epsilon \Longrightarrow FIRST(\beta) \subseteq FIRST(\alpha)$
$X_1, X_2, \ldots, X_k$, 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 $\forall j < k \quad \epsilon \in FIRST(X_j)$

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

Definition of FOLLOW

Grammar:

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

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

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

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

```
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:

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

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 EOF.
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: $A \longrightarrow a$ $\quad B \longrightarrow b$ $\quad S \longrightarrow A\ S\ B$ $\quad S \longrightarrow \epsilon$

$First(S)$	$=$	{ a, ϵ }	$Follow(S)$	$=$	{ b, EOF }
$First(A)$	$=$	{ a }	$Follow(A)$	$=$	{ a, b }
$First(B)$	$=$	{ b }	$Follow(B)$	$=$	{ b, 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:

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

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

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

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

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

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

Removing Left Recursion: Another Example

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

$$\Downarrow$$

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

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

Parsing with LL(1) Grammars: Example 2

Nonterminal	Input Symbol		
	id	+	EOF
E	$E \longrightarrow$ id E'		
E'		$E' \longrightarrow$ + $E\ E'$	$E' \longrightarrow \epsilon$

Stack	Input Stream	Rule	Derivation
$\$E$	id + id$	$E \longrightarrow$ id E'	$E \Longrightarrow$ id E'
$\$E'$ id	id + id$		
$\$E'$	+ id$	$E' \longrightarrow$ + id E'	$\Longrightarrow$ 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 $\alpha \stackrel{*}{\Longrightarrow} \epsilon$ 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).