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 above tree is used later 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* (like DFA) can be constructed to efficiently 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:

$\surd$ $\{a, b, c\}$
$\surd$ $\{\epsilon, a, b, aa, ab, ba, bb, \ldots\}$
$\surd$ $\{(ab)^n \mid n \geq 0\}$
$\times$ $\{a^n b^n \mid n \geq 0\}$

Grammars

Notation where recursion is explicit.
Examples:

- $\{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \ldots\}$:

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

Notational shorthand:

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

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

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

- $\{w \mid \textit{no. of } \mathtt{a}\textit{'s in } w = \textit{no. of } \mathtt{b}\textit{'s in } w\}$

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: nonterminal symbol that represents the set of all strings in the language.

Example

$$\begin{array}{rcl} E & \longrightarrow & E + E \\ E & \longrightarrow & E - E \\ E & \longrightarrow & E * E \\ E & \longrightarrow & E \;/\; E \\ E & \longrightarrow & (\;E\;) \\ E & \longrightarrow & \mathsf{id} \end{array}$$

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

Context-free Grammars

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

Context-Free Vs Other Types of Grammars

- Context-free grammar (CFG): Productions of the form $NT \longrightarrow [NT|T]*$
- Context-sensitive grammar (CSG): Productions of the form $[t|NT] * NT[t|NT]* \longrightarrow [t|NT]*$
- Unrestricted grammar: Productions of the form $[t|NT]* \longrightarrow [t|NT]*$

Examples of Non-Context-Free Languages

- Checking that variables are declared before use. If we simplify and abstract the problem, we see that it amounts to recognizing strings of the form wsw
- Checking whether the number of actual and formal parameters match. Abstracts to recognizing strings of the form $a^n b^m c^n d^m$
- In both cases, the rules are not enforced in grammar, but deferred to type-checking phase
- Note: Strings of the form wsw^R and $a^n b^n c^m d^m$ can be described by a CFG

What types of Grammars Describe These Languages?

- Strings of 0's and 1's of the form xx
- Strings of 0's and 1's in which 011 doesn't occur
- Strings of 0's and 1's in which each 0 is immediately followed by a 1
- Strings of 0's and 1's with equal number of 0's and 1's.

Language Generated by Grammars, Equivalence of Grammars

- How to show that a grammar G generates a language $\mathcal{M}$? Show that
  - $\forall s \in \mathcal{M}$, show that $s \in \mathcal{L}(G)$
  - $\forall s \in \mathcal{L}(G)$, show that $s \in \mathcal{M}$
- How to establish that two grammars G_1 and G_2 are equivalent?
  Show that $\mathcal{L}(G_1) = \mathcal{L}(G_2)$

Grammar Examples

$$S \longrightarrow 0S1S|1S0S|\epsilon$$

What is the language generated by this grammar?

Grammar Examples

$$\begin{aligned} S &\longrightarrow 0A|1B|\epsilon \\ A &\longrightarrow 0AA|1S \\ B &\longrightarrow 1BB|0S \end{aligned}$$

What is the language generated by this grammar?

The Two Sides of Grammars

Specify a set of strings in a language.
Recognize strings in a given language:

- Is a given string x in the language?

  Yes, if we can construct a *derivation* for x
- Example: Is $\mathsf{id} + \mathsf{id} \in \mathcal{L}(E)$?

$$\begin{aligned} \mathsf{id} + \mathsf{id} &\Longleftarrow E + \mathsf{id} \\ &\Longleftarrow E + E \\ &\Longleftarrow E \end{aligned}$$

Derivations

$$\boxed{\textbf{Grammar:}\quad \begin{array}{lcl} E & \longrightarrow & E + E \\ E & \longrightarrow & \mathsf{id} \end{array}}$$

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

$$\begin{array}{lcl} E & \Longrightarrow & E + E \\ & \Longrightarrow & E + \mathsf{id} \\ & \Longrightarrow & \mathsf{id} + \mathsf{id} \end{array}$$

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

Derivations

- Rightmost derivation: Rightmost nonterminal is replaced first:

$$\begin{array}{lcl} E & \Longrightarrow & E + E \\ & \Longrightarrow & E + \mathsf{id} \\ & \Longrightarrow & \mathsf{id} + \mathsf{id} \end{array}$$

  Written as $E \stackrel{*}{\Longrightarrow}_{rm} \mathsf{id} + \mathsf{id}$

- Leftmost derivation: Leftmost nonterminal is replaced first:

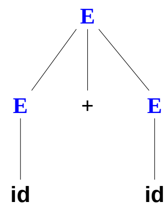
$$\begin{array}{lcl} E & \Longrightarrow & E + E \\ & \Longrightarrow & \mathsf{id} + E \\ & \Longrightarrow & \mathsf{id} + \mathsf{id} \end{array}$$

  Written as $E \stackrel{*}{\Longrightarrow}_{lm} \mathsf{id} + \mathsf{id}$

Parse Trees

Graphical Representation of Derivations

$$\begin{array}{lcl} E & \Longrightarrow & E + E \\ & \Longrightarrow & \mathsf{id} + E \\ & \Longrightarrow & \mathsf{id} + \mathsf{id} \end{array}$$

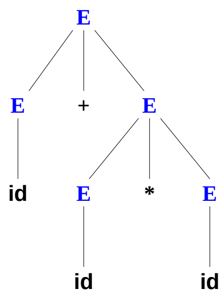


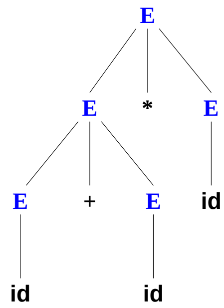
$$\begin{array}{lcl} E & \Longrightarrow & E + E \\ & \Longrightarrow & E + \mathsf{id} \\ & \Longrightarrow & \mathsf{id} + \mathsf{id} \end{array}$$

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

Ambiguity

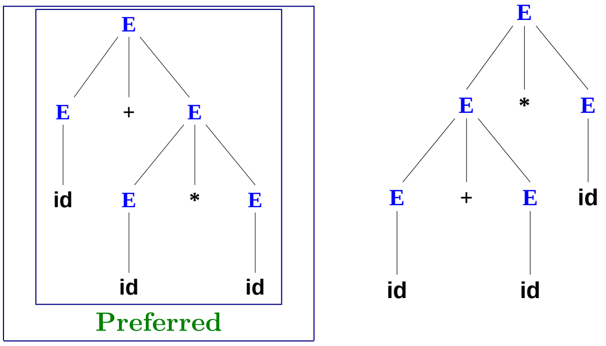
A Grammar is *ambiguous* if there are multiple parse trees for the same sentence.
Example: $\mathsf{id} + \mathsf{id} * \mathsf{id}$





Disambiguition

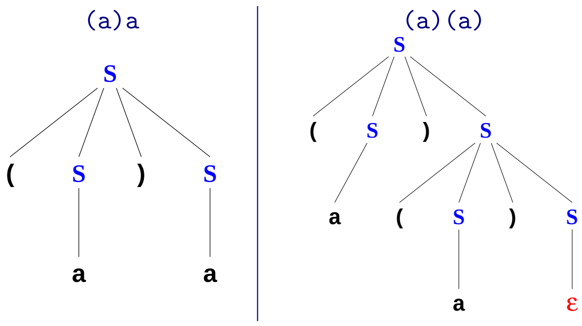
Express Preference for one parse tree over others.
Example: id + id ∗ id
The usual precedence of ∗ over + means:



Parsing

Construct a parse tree for a given string.

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



A Procedure for Parsing

Grammar: $S \longrightarrow a$

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

Predictive Parsing

Grammar:

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

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

Predictive Parsing (Contd.)

Grammar:

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

```
procedure parse_S() {
    switch (input_token) {
        case TOKEN_OPEN_PAREN:    /* Production 1 */
            consume(TOKEN_OPEN_PAREN);
            parse_S();
            consume(TOKEN_CLOSE_PAREN);
            parse_S();
            return;
```

Predictive Parsing (contd.)

Grammar:

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

```
        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

Grammar cannot be left-recursive

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

```
procedure 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

$$\begin{array}{lcl} A & \longrightarrow & A\ a \\ A & \longrightarrow & b \end{array}$$

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

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

Removing Left Recursion

More generally,

$$\begin{array}{lcl} A & \longrightarrow & A\alpha_1 | \cdots | A\alpha_m \\ A & \longrightarrow & \beta_1 | \cdots | \beta_n \end{array}$$

Can be transformed into

$$\begin{array}{lcl} A & \longrightarrow & \beta_1 A' | \cdots | \beta_n A' \\ A' & \longrightarrow & \alpha_1 A' | \cdots | \alpha_m A' | \epsilon \end{array}$$

Removing Left Recursion: An Example

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

$$\Downarrow$$

$$\begin{array}{lcl} E & \longrightarrow & \mathsf{id}\ E' \\ E' & \longrightarrow & +\ E\ E' \\ E' & \longrightarrow & \epsilon \end{array}$$

Predictive Parsing: Restrictions

May not be able to choose a *unique* production

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

Left-factoring can help:

$$
\begin{array}{rcl}
S & \longrightarrow & a\ B\ d \\
B & \longrightarrow & bC \\
C & \longrightarrow & c|\epsilon
\end{array}
$$

Predictive Parsing: Restrictions

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

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

Recursive Descent Parsing

Grammar: $\begin{array}{rcl} S & \longrightarrow & a\ B\ d \\ B & \longrightarrow & b \\ B & \longrightarrow & bc \end{array}$

```
procedure parse_B() {
    switch (input_token) {
        case TOKEN_b:   /* Production 2 */
            consume(TOKEN_b);
            return;
        case TOKEN_b:   /* Production 3 */
            consume(TOKEN_b);
            consume(TOKEN_c);
            return;
}}
```

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

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

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		$B \longrightarrow b$	

Table-driven Parsing Algorithm

```
stack initialized to EOF.
while (stack is not empty) {
    X = top(stack);
    if (X is a terminal symbol)
        consume(X);
    else /* X is a nonterminal */
        if (M[X, input_token] = X ⟶ Y1, Y2, ..., Yk) {
            pop(stack);
            for i = k downto 1 do
                push(stack, Yi);
        }
        else /* Syntax Error */
}
```

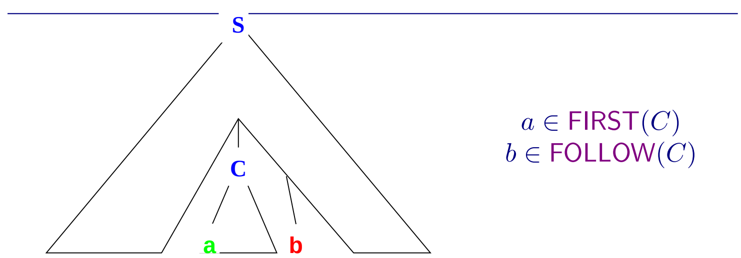
FIRST and FOLLOW

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

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

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

  FOLLOW(S) = $\{), \mathsf{EOF}\}$



FIRST and FOLLOW

$\mathit{FIRST}(X)$: First terminal in some α such that $X \stackrel{*}{\Longrightarrow} \alpha$.

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

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

$$\begin{array}{lcl} \mathit{First}(S) & = & \{\ \mathsf{a},\ \epsilon\ \} \\ \mathit{First}(A) & = & \{\ \mathsf{a}\ \} \\ \mathit{First}(B) & = & \{\ \mathsf{b}\ \} \end{array} \qquad \begin{array}{lcl} \mathit{Follow}(S) & = & \{\ \mathsf{b},\ \mathsf{EOF}\ \} \\ \mathit{Follow}(A) & = & \{\ \mathsf{a},\ \mathsf{b}\ \} \\ \mathit{Follow}(B) & = & \{\ \mathsf{b},\ \mathsf{EOF}\ \} \end{array}$$

Definition of FIRST

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

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

$\alpha =$	Property of $\mathit{FIRST}(\alpha)$
a, a terminal	$a \in \mathit{FIRST}(\alpha)$
A, a nonterminal	$A \longrightarrow \epsilon \in G \Longrightarrow \epsilon \in \mathit{FIRST}(\alpha)$ $A \longrightarrow \beta \in G,\ \beta \neq \epsilon \Longrightarrow \mathit{FIRST}(\beta) \subseteq \mathit{FIRST}(\alpha)$
$X_1X_2\cdots X_k$, a string of terminals and nonterminals	$\mathit{FIRST}(X_1) - \{\epsilon\} \subseteq \mathit{FIRST}(\alpha)$ $\mathit{FIRST}(X_i) \subseteq \mathit{FIRST}(\alpha)$ if $\forall j < i \quad \epsilon \in \mathit{FIRST}(X_j)$ $\epsilon \in \mathit{FIRST}(\alpha)$ if $\forall j < k \quad \epsilon \in \mathit{FIRST}(X_j)$

Definition of FOLLOW

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

$\mathit{FOLLOW}(A)$ is the smallest set such that

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

A Procedure to Construct Parsing Tables

procedure $\mathit{table\_construct}(G)$ {
  for each $A \longrightarrow \alpha \in G$ {
    for each $a \in \mathit{FIRST}(\alpha)$ such that $a \neq \epsilon$
      add $A \longrightarrow \alpha$ to $M[A, a]$;
    if $\epsilon \in \mathit{FIRST}(\alpha)$
      for each $b \in \mathit{FOLLOW}(A)$
        add $A \longrightarrow \alpha$ to $M[A, b]$;
}}

LL(1) Grammars

Grammars for which the parsing table constructed earlier has no multiple entries.

$$
\begin{array}{lcl}
E & \longrightarrow & \mathsf{id}\ E' \\
E' & \longrightarrow & +\ E\ E' \\
E' & \longrightarrow & \epsilon
\end{array}
$$

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

Parsing with LL(1) Grammars

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

$\$E$	id + id\$	$E \Longrightarrow$	idE'
$\$E'$id	id + id\$		
$\$E'$	+ id\$	$\Longrightarrow$	id+EE'
$\$E'E+$	+ id\$		
$\$E'E$	id\$	$\Longrightarrow$	id+id$E'E'$
$\$E'E'$id	id\$		
$\$E'E'$	\$	$\Longrightarrow$	id+idE'
$\$E'$	\$	$\Longrightarrow$	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. $\mathit{FIRST}(\alpha) \cap \mathit{FIRST}(\beta) = \{\ \}$, and
2. if $\alpha \stackrel{*}{\Longrightarrow} \epsilon$ then $\mathit{FIRST}(\beta) \cap \mathit{FOLLOW}(A) = \{\ \}$.

Corollary: No Ambiguous Grammar is LL(1).

Leftmost and Rightmost Derivations

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

Derivations for id + id:

$$
\begin{array}{lcl|lcl}
E & \Longrightarrow & E+T & E & \Longrightarrow & E+T \\
 & \Longrightarrow & T+T & & \Longrightarrow & E+\mathsf{id} \\
 & \Longrightarrow & \mathsf{id}+T & & \Longrightarrow & T+\mathsf{id} \\
 & \Longrightarrow & \mathsf{id}+\mathsf{id} & & \Longrightarrow & \mathsf{id}+\mathsf{id} \\
 & \text{LEFTMOST} & & & \text{RIGHTMOST} &
\end{array}
$$

Bottom-up Parsing

Given a stream of tokens w, *reduce* it to the start symbol.

$$\begin{array}{lcl} \hline E & \longrightarrow & E{+}T \\ E & \longrightarrow & T \\ T & \longrightarrow & \mathsf{id} \\ \hline \end{array}$$

Parse input stream: $\mathsf{id} + \mathsf{id}$:

$$\begin{array}{c} \mathsf{id} + \mathsf{id} \\ T + \mathsf{id} \\ E + \mathsf{id} \\ E + T \\ E \end{array}$$

Reduction $\equiv$ Derivation^{-1}.

Handles

Informally, a "handle" of a string is a substring that matches the right side of a production, and whose reduction to the nonterminal on the left hand side of the production represents one step along the reverse rightmost derivation.

Handles

A structure that furnishes a means to perform reductions.

$$\begin{array}{lcl} \hline E & \longrightarrow & E{+}T \\ E & \longrightarrow & T \\ T & \longrightarrow & \mathsf{id} \\ \hline \end{array}$$

Parse input stream: $\mathsf{id} + \mathsf{id}$:

$$\begin{array}{c} \boxed{\mathsf{id}} + \mathsf{id} \\ \boxed{T} + \mathsf{id} \\ E + \boxed{\mathsf{id}} \\ \boxed{E + T} \\ E \end{array}$$

Handles

Handles are substrings of sentential forms:

1. A substring that matches the right hand side of a production
2. Reduction using that rule can lead to the start symbol

$$\begin{array}{lcl} E & \Longrightarrow & \boxed{E + T} \\ & \Longrightarrow & E + \boxed{\mathsf{id}} \\ & \Longrightarrow & \boxed{T} + \mathsf{id} \\ & \Longrightarrow & \boxed{\mathsf{id}} + \mathsf{id} \end{array}$$

Handle Pruning: replace handle by corresponding LHS.

Shift-Reduce Parsing

Bottom-up parsing.

- **Shift:** Construct leftmost handle on top of stack

- **Reduce:** Identify handle and replace by corresponding RHS
- **Accept:** Continue until string is reduced to start symbol and input token stream is empty
- **Error:** Signal parse error if no handle is found.

Implementing Shift-Reduce Parsers

- Stack to hold grammar symbols (corresponding to tokens seen thus far).
- Input stream of yet-to-be-seen tokens.
- Handles appear on top of stack.
- Stack is initially empty (denoted by \$).
- Parse is successful if stack contains only the start symbol when the input stream ends.

Shift-Reduce Parsing: An Example

$$\begin{array}{ccl} S & \longrightarrow & aABe \\ A & \longrightarrow & Abc|b \\ B & \longrightarrow & d \end{array}$$

To parse: $a\ b\ b\ c\ d\ e$

Shift-Reduce Parsing: An Example

$$\begin{array}{ccl} E & \longrightarrow & E{+}T \\ E & \longrightarrow & T \\ T & \longrightarrow & \mathsf{id} \end{array}$$

STACK	INPUT STREAM	ACTION
\$	id + id \$	shift
\$ id	+ id \$	reduce by $T \longrightarrow$ id
\$ T	+ id \$	reduce by $E \longrightarrow T$
\$ E	+ id \$	shift
\$ E +	id \$	shift
\$ E + id	\$	reduce by $T \longrightarrow$ id
\$ E + T	\$	reduce by $E \longrightarrow E{+}T$
\$ E	\$	**ACCEPT**

More on Handles

Handle: Let $S \Longrightarrow^*_{rm} \alpha A w \Longrightarrow_{rm} \alpha\beta w$.
Then $A \longrightarrow \beta$ is a handle for $\alpha\beta w$ at the position imeediately following α.

Notes:

- For unambiguous grammars, every right-sentential form has a unique handle.
- In shift-reduce parsing, handles always appear on top of stack, i.e., $\alpha\beta$ is in the stack (with β at top), and w is unread input.

Identification of Handles and Relationship to Conflicts

Case 1: With $\alpha\beta$ on stack, don't know if we hanve a handle on top of stack, or we need to shift some more input to get βx which is a handle.

- Shift-reduce conflict
- Example: if-then-else

Case 2: With $\alpha\beta_1\beta_2$ on stack, don't know if $A \longrightarrow \beta_2$ is the handle, or $B \longrightarrow \beta_1\beta_2$ is the handle

- Reduce-reduce conflict
- Example: $E \longrightarrow E - E| - E|id$

Viable Prefix

Prefix of a right-sentential form that does not continue beyond the rightmost handle.
With $\alpha\beta w$ example of the previous slides, a viable prefix is something of the form $\alpha\beta_1$ where $\beta = \beta_1\beta_2$

LR Parsing

- Stack contents as $s_0X_1s_1X_2\cdots X_ms_m$
- Its actions are driven by two tables, *action* and *goto*

Parser Configuration: $(\underbrace{s_0X_1s_1X_2\cdots X_ms_m}_{\text{stack}},\ \underbrace{a_ia_{i+1}\cdots a_n\$}_{\text{unconsumed input}})$

$action[s_m, a_i]$ can be:

- shift s: new config is $(s_0X_1s_1X_2\cdots X_ms_ma_is, a_{i+1}\cdots a_n\$)$
- reduce $A \longrightarrow \beta$: Let $|\beta| = r$, $goto[s_{m-r}, A] = s$: new config is $(s_0X_1s_1X_2\cdots X_{m-r}s_{m-r}As, a_ia_{i+1}\cdots a_n\$)$
- error: perform recovery actions
- accept: Done parsing

LR Parsing

- *action* and *goto* depend only on the state at the top of the stack, not on all of the stack contents
  - The s_i states compactly summarize the "relevant" stack content that is at the top of the stack.
- You can think of *goto* as the action taken by the parser on "consuming" (and shifting) nonterminals
  - similar to the shift action in the *action* table, except that the transition is on a nonterminal rather than a terminal
- The *action* and *goto* tables define the transitions of an FSA that accepts RHS of productions!

Example of LR Parsing Table and its Use

- See Text book Algorithm 4.7: (follows directly from description of LR parsing actions 2 slides earlier)
- See expression grammar (Example 4.33), its associated parsing table in Fig 4.31, and the use of the table to parse $id * id + id$ (Fig 4.32)

LR Versus LL Parsing

Intuitively:

- LL parser needs to guess the production based on the first symbol (or first few symbols) on the RHS of a production
- LR parser needs to guess the production *after* seeing all of the RHS

Both types of parsers can use next k input symbols as look-ahead symbols (LL(k) and LR(k) parsers)

- Implication: $LL(k) \subset LR(k)$

How to Construct LR Parsing Table?

Key idea: Construct an FSA to recognize RHS of productions

- States of FSA remember which parts of RHS have been seen already.
- We use " · " to separate seen and unseen parts of RHS

LR(0) item: A production with " · " somewhere on the RHS. Intuitively,

▷ grammar symbols before the " · " are on stack;

▷ grammar symbols after the " · " represent symbols in the input stream.

$$I_0: \begin{array}{l} E' \longrightarrow \cdot E \\ E \longrightarrow \cdot E+T \\ E \longrightarrow \cdot T \\ T \longrightarrow \cdot \text{id} \end{array}$$

How to Construct LR Parsing Table?

- If there is no way to distinguish between two different productions at some point during parsing, then the same state should represent both.
  - *Closure* operation: If a state s includes LR(0) item $A \longrightarrow \alpha \cdot B\beta$, and there is a production $B \longrightarrow \gamma$, then s should include $B \longrightarrow \cdot \gamma$
  - *goto* operation: For a set I of items, $goto[I, X]$ is the closure of all items $A \longrightarrow \alpha X \cdot \beta$ for each $A \longrightarrow \alpha \cdot X\beta$ in I

Item set: A set of items that is closed under the *closure* operation, corresponds to a state of the parser.

Constructing Simple LR (SLR) Parsing Tables

Step 1: Construct LR(0) items (Item set construction)

Step 2: Construct a DFA for recognizing items

Step 3: Define *action* and *goto* based from the DFA

Item Set Construction

1. Augment the grammar with a rule $S' \longrightarrow S$, and make S' the new start symbol

2. Start with initial set I_0 corresponding to the item $S' \longrightarrow \cdot S$
3. apply *closure* operation on I_0.
4. For each item set I and grammar symbol X, add $goto[I, X]$ to the set of items
5. Repeat previous step until no new item sets are generated.

Item Set Construction

$E' \longrightarrow E$ $\quad E \longrightarrow E + T \mid T$ $\quad T \longrightarrow T * F \mid F$ $\quad F \longrightarrow (E) \mid id$

$I_0 : E' \longrightarrow \cdot E$

$I_3 : T \longrightarrow F \cdot$

$I_4 : F \longrightarrow (\cdot E)$

$I_1 : E' \longrightarrow E \cdot$

$I_2 : E \longrightarrow T \cdot$

$I_5 : F \longrightarrow id \cdot$

Item Set Construction (Continued)

$E' \longrightarrow E$ $\quad E \longrightarrow E + T \mid T$ $\quad T \longrightarrow T * F \mid F$ $\quad F \longrightarrow (E) \mid id$

$I_6 : E \longrightarrow E + \cdot T$

$I_8 : F \longrightarrow (E \cdot)$

$I_9 : E \longrightarrow E + T \cdot$

$I_7 : T \longrightarrow T * \cdot F$

$I_{10} : T \longrightarrow T * F \cdot$

$I_{11} : F \longrightarrow (E) \cdot$

Item Sets for the Example

I_0: $E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot\mathbf{id}$

I_1: $E' \to E \cdot$
$E \to E \cdot + T$

I_2: $E \to T \cdot$
$T \to T \cdot * F$

I_3: $T \to F \cdot$

I_4: $F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot\mathbf{id}$

I_5: $F \to \mathbf{id} \cdot$

I_6: $E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot\mathbf{id}$

I_7: $T \to T * \cdot F$
$F \to \cdot(E)$
$F \to \cdot\mathbf{id}$

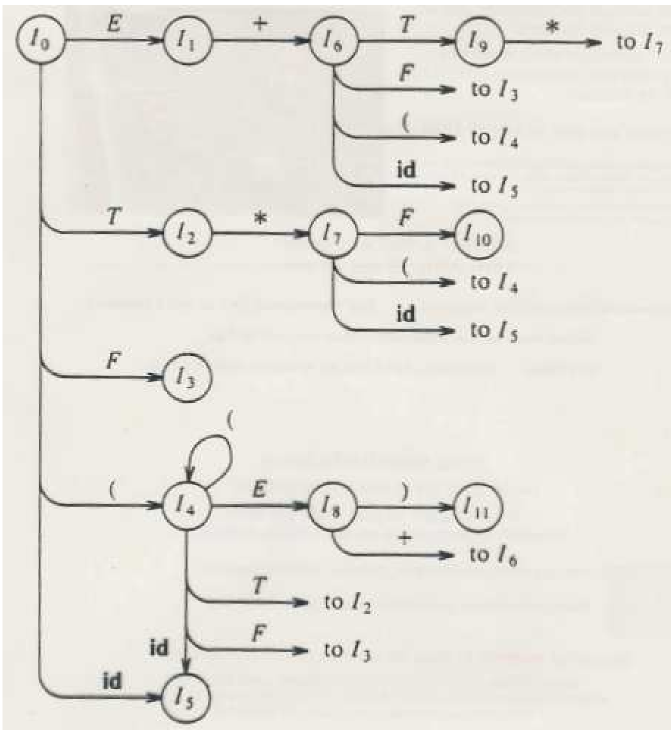
I_8: $F \to (E \cdot)$
$E \to E \cdot + T$

I_9: $E \to E + T \cdot$
$T \to T \cdot * F$

I_{10}: $T \to T * F \cdot$

I_{11}: $F \to (E) \cdot$

Constructing DFA to Recognize Viable Prefixes



SLR(1) Parse Table for the Example Grammar

STATE	*action*						*goto*		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Define *action* and *goto* tables

- Let $I_0, I_1, \ldots, I_n$ be the item sets constructed before
- Define *action* as follows
  - If $A \longrightarrow \alpha \cdot a\beta$ is in I_i and there is a DFA transition to I_j from I_i on symbol a then $action[i, a] =$ "shift j"
  - If $A \longrightarrow \alpha\,\cdot$ is in I_i then $action[i, a] =$ "reduce $A \longrightarrow \alpha$" for every $a \in FOLLOW(A)$
  - If $S' \longrightarrow S\,\cdot$ is in I_i then $action[I_i, \$] =$ "accept"
- If any conflicts arise in the above procedure, then the grammar is *not* SLR(1).
- *goto* transition for LR parsing defined directly from the DFA transitions.
- All undefined entries in the table are filled with "error"

Deficiencies of SLR Parsing

SLR(1) treats all occurrences of a RHS on stack as identical.
Only a few of these reductions may lead to a successful parse.
Example:

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

$I_0 = \{[S' \rightarrow \cdot S], [S \rightarrow \cdot A\mathsf{a}A\mathsf{b}], [S \rightarrow \cdot B\mathsf{b}B\mathsf{a}], [A \rightarrow \cdot], [B \rightarrow \cdot]\}$.
Since $\mathit{FOLLOW}(A) = \mathit{FOLLOW}(B)$, we have reduce/reduce conflict in state 0.

LR(1) Item Sets

Construct LR(1) items of the form $A \longrightarrow \alpha \cdot \beta, \mathsf{a}$, which means:

The production $A \longrightarrow \alpha\beta$ can be applied when the next token on input stream is a.

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

An example LR(1) item set:
$I_0 = \{[S' \to \cdot\ S, \$], [S \to \cdot\ A\mathsf{a}A\mathsf{b}, \$], [S \to \cdot\ B\mathsf{b}B\mathsf{a}, \$],$
$[A \to \cdot\ , \mathsf{a}], [B \to \cdot\ , \mathsf{b}]\}.$

LR(1) and LALR(1) Parsing

LR(1) parsing: Parse tables built using LR(1) item sets.

LALR(1) parsing: *Look Ahead* LR(1)

Merge LR(1) item sets; then build parsing table.

Typically, LALR(1) parsing tables are much smaller than LR(1) parsing table.

YACC

Yet **A**nother **C**ompiler **C**ompiler:
LALR(1) parser generator.

- Grammar rules written in a specification (`.y`) file, analogous to the regular definitions in a `lex` specification file.
- Yacc translates the specifications into a parsing function `yyparse()`.

$$\texttt{spec.y} \xrightarrow{\texttt{yacc}} \texttt{spec.tab.c}$$

- `yyparse()` calls `yylex()` whenever input tokens need to be consumed.
- `bison`: GNU variant of `yacc`.

Using Yacc

```
%{
  ... C headers (#include)
%}

... Yacc declarations:
       %token ...
       %union{...}
       precedences
%%
... Grammar rules with actions:

Expr:  Expr TOK_PLUS Expr
    |  Expr TOK_MINUS Expr
    ;
%%
... C support functions
```


YACC

Yet **A**nother **C**ompiler **C**ompiler:
LALR(1) parser generator.

- Grammar rules written in a specification (`.y`) file, analogous to the regular definitions in a `lex` specification file.
- Yacc translates the specifications into a parsing function `yyparse()`.

$$\texttt{spec.y} \xrightarrow{\texttt{yacc}} \texttt{spec.tab.c}$$

- `yyparse()` calls `yylex()` whenever input tokens need to be consumed.
- `bison`: GNU variant of `yacc`.

Using Yacc

```
%{
  ... C headers (#include)
%}

... Yacc declarations:
       %token ...
       %union{...}
       precedences
%%
... Grammar rules with actions:

Expr:  Expr TOK_PLUS Expr
    |  Expr TOK_MINUS Expr
    ;
%%
... C support functions
```

Conflicts and Resolution

- Operator precedence works well for resolving conflicts that involve operators
  - But use it with care – only when they make sense, not for the sole purpose of removing conflict reports
- Shift-reduce conflicts: Bison favors shift
  - Except for the dangling-else problem, this strategy does not ever seem to work, so don't rely on it.

Reduce-Reduce Conflicts

```
sequence: /* empty */
          { printf ("empty sequence\n"); }
        | maybeword
        | sequence word
          { printf ("added word %s\n", $2); };

maybeword: /* empty */
           { printf ("empty maybeword\n"); }
        | word
          { printf ("single word %s\n", $1); };
```

In general, grammar needs to be rewritten to eliminate conflicts.

Sample Bison File: Postfix Calculator

```
input:    /* empty */
        | input line
;
line:     '\n'
        | exp '\n'      { printf ("\t%.10g\n", $1); }
;
exp:      NUM           { $$ = $1;           }
        | exp exp '+'   { $$ = $1 + $2;      }
        | exp exp '-'   { $$ = $1 - $2;      }
        | exp exp '*'   { $$ = $1 * $2;      }
        | exp exp '/'   { $$ = $1 / $2;      }
         /* Exponentiation */
        | exp exp '^'   { $$ = pow ($1, $2); }
         /* Unary minus    */
        | exp 'n'       { $$ = -$1;          };
%%
```

Infix Calculator

```
%{
#define YYSTYPE double
#include <math.h>
#include <stdio.h>
int yylex (void);
void yyerror (char const *);
%}
/* Bison Declarations */
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG     /* negation--unary minus */
%right '^'    /* exponentiation */
```

Infix Calculator (Continued)

```
%% /* The grammar follows.  */
input:    /* empty */
        | input line
;
line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;
exp:      NUM                { $$ = $1;         }
        | exp '+' exp        { $$ = $1 + $3;    }
        | exp '-' exp        { $$ = $1 - $3;    }
        | exp '*' exp        { $$ = $1 * $3;    }
        | exp '/' exp        { $$ = $1 / $3;    }
        | '-' exp  %prec NEG { $$ = -$2;        }
        | exp '^' exp        { $$ = pow ($1, $3); }
        | '(' exp ')'        { $$ = $2;         }
;
%%
```

Error Recovery

```
line:     '\n'
        | exp '\n'   { printf ("\t%.10g\n", $1); }
        | error '\n' { yyerrok;                  };
```

- Pop stack contents to expose a state where `error` token is acceptable
- Shift error token onto the stack
- Discard input until reaching a token that can follow this error token

Error recovery strategies are never perfect — some times they lead to cascading errors, unless carefully designed.

Left Versus Right Recursion

```
expseq1:  exp | expseq1 ',' exp;
```

is a left-recursive definition of a sequence of `exp`'s, whereas

```
expseq1:  exp | exp ',' expseq1;
```

is a right-recursive definition

- Left-recursive definitions are no-no for LL parsing, but yes-yes for LR parsing
- Right-recursive definition is bad for LR parsing as it needs to shift entire list on stack before any reduction — increases stack usage