Parser Generation

- **Main Problem**: given a grammar $G$, how to build a top-down parser or a bottom-up parser for it?
- **Parser**: a program that, given a sentence, reconstructs a derivation for that sentence — if done successfully, it "recognize" the sentence
- All parsers read their input *left-to-right*, but construct parse tree differently.
- **Bottom-up parsers**: construct the tree from leaves to root
  - shift-reduce, LR, SLR, LALR, operator precedence
- **Top-down parsers**: construct the tree from root to leaves
  - recursive descent, predictive parsing, LL(1)

Bottom-Up Parsing

- Construct parse tree "bottom-up" — from leaves to the root
- Bottom-up parsing always constructs right-most derivation
- Important parsing algorithms: *shift-reduce*, LR parsing
- LR parser components: input, stack (strings of grammar symbols and states), driver routine, parsing tables.

LR Parsing

- A sequence of new *state* symbols $s_0, s_1, s_2, \ldots, s_m$ — each state summarize the information contained in the stack below it.
- Parsing configurations: *(stack, remaining input)* written as
  $$ (s_0X_1s_1X_2s_2\ldots X_ms_m, \ a_1a_1+1a_1+2\ldots a_n\delta) $$
  next "move" is determined by $s_m$ and $a_i$
- Parsing tables:
  - **ACTION**[$s,a$] and **GOTO**[$s,X$]

  **Table A**
  
<table>
<thead>
<tr>
<th>$s$ : state</th>
<th>$a$ : terminal</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) shift $s_k$</td>
<td>(2) reduce $A \rightarrow \beta$</td>
</tr>
<tr>
<td>(3) accept</td>
<td>(4) error</td>
</tr>
</tbody>
</table>

  **Table G**
  
<table>
<thead>
<tr>
<th>$s$ : state</th>
<th>$X$ : non-terminal</th>
</tr>
</thead>
<tbody>
<tr>
<td>its entries are <em>states</em></td>
<td></td>
</tr>
</tbody>
</table>

Constructing LR Parser

*How to construct the parsing table ACTION and GOTO?*

- **Basic idea**: first construct DFA to recognize handles, then use DFA to construct the parsing tables! different parsing table yield different LR parsers *SLR(1)*, *LR(1)*, or *LALR(1)*
- **Augmented grammar** for context-free grammar $G = (T,N,P,S)$ is defined as $G' = (T \cup \{S'\}, N \cup \{S' \rightarrow S\}, S')$ — adding non-terminal $S'$ and the production $S' \rightarrow S$, and $S'$ is the new start symbol. When $S' \rightarrow S$ is reduced, parser accepts.
- **LR(0) item** for productions of a context-free grammar $G$ — is a production with *dot* at some position in the r.h.s.
  - For $A \rightarrow XYZ$, its items are $A \rightarrow .XYZ \rightarrow X.YZ$ $A \rightarrow X.YZ$
  - For $A \rightarrow \epsilon$, its items are just $A \rightarrow .$
LR(0) items and LR(0) DFA

- Informally, item \( A \rightarrow X. YZ \) means a string derivable from \( X \) has been seen, and one from \( YZ \) is expected. LR(0) items are used as state names for LR(0) DFA or LR(0) NFA that recognizes viable prefixes.

- Viable prefixes of a CFG are prefixes of right-sentential forms with no symbols to right of the handle; we can always add terminals on right to form a right-sentential form.

- Two ways to construct the LR(0) DFA:
  1. first construct LR(0) NFA and then convert it to a DFA!
  2. construct the LR(0) DFA directly!

- From LR(0) DFA to the Parsing Table

Example: LR(0) Items

**CFG Grammar:**

\[
\begin{align*}
E & \rightarrow E + T | T \\
T & \rightarrow T * F | F \\
F & \rightarrow ( E ) | id
\end{align*}
\]

**Augmented Grammar:**

\[
\begin{align*}
E' & \rightarrow E \\
E & \rightarrow E + T | T \\
T & \rightarrow T * F | F \\
F & \rightarrow ( E ) | id
\end{align*}
\]

**LR(0) terms:**

\[
\begin{align*}
E' & \rightarrow . E \\
T & \rightarrow . T \\
F & \rightarrow ( E ) | id \\
E & \rightarrow E + . T \\
E & \rightarrow E * . T \\
E & \rightarrow . T \\
E & \rightarrow . F
\end{align*}
\]

From LR(0) NFA to LR(0) DFA

- Construct LR(0) NFA with all LR(0) items of \( G \) as states, connect states by moving the dot. Final states are those with dots at the end.

1. for each item \( A \rightarrow \alpha . X \beta \)

   \[
   \begin{array}{c}
   A \rightarrow \alpha . X \beta \\
   A \rightarrow \alpha X . \beta
   \end{array}
   \]

2. for each pair \( A \rightarrow \alpha . B \beta , B \rightarrow . \gamma \)

   (expect to see a string derivable from \( \gamma \))

   \[
   \begin{array}{c}
   A \rightarrow \alpha . B \beta \\
   B \rightarrow . \gamma
   \end{array}
   \]

- Convert NFA to DFA using subset construction algorithm.

- The states of the resulting LR(0) DFA --- \( C = \{ \Gamma_1, \Gamma_2, \ldots, \Gamma_n \} \) are called canonical LR(0) collection for grammar \( G' \)

- Disadvantage: the NFA is often huge, and converting from NFA to DFA is tedious and time-consuming.

Building LR(0) DFA Directly

- Instead of building DFA from NFA, we can build the LR(0) DFA directly.

- Given a set of LR(0) items \( I \), \( \text{CLOSURE}(I) \) is defined as

\[
\text{repeat} \\
\text{for each item } A \rightarrow \alpha . B \beta \text{ in } I \text{ and each production } B \rightarrow \gamma \\
\text{add } B \rightarrow . \gamma \text{ to } I, \text{ if it's not in } I \\
\text{until } I \text{ does not change}
\]

- \( \text{GOTO}(I,X) \) is defined as

\[
\text{CLOSURE}(\text{all items } A \rightarrow \alpha . X \beta \text{ for each } A \rightarrow \alpha . X \beta \text{ in } I)
\]

- Canonical LR(0) collection is computed by the following procedure:

\[
\begin{align*}
\Gamma_0 &= \text{CLOSURE}([S' \rightarrow . S]) \text{ and } C = \{ \Gamma_0 \} \\
\text{repeat} \\
&\text{for each } I \in C \text{ and grammar symbol } X \\
&\text{T} = \text{GOTO}(I,X); \text{ if } T \neq \emptyset \text{ and } T \subseteq C \text{ then } C = C \cup \{ T \}; \\
&\text{until } C \text{ does not change}
\end{align*}
\]

Resulting LR(0) DFA: \( C \) is the set of states; \( \text{GOTO} \) is the transition table.
Constructing SLR(1) Parsing Table

- From the LR(0) DFA, we can construct the parsing table — SLR(1) parsing table. The parser based on SLR(1) parsing table is called SLR(1) parser. The SLR(1) grammars are those whose SLR(1) parsing table does not contain any conflicts.

- Algorithm — use C = \{I_0, \ldots, I_n\}, GOTO, FOLLOW:
  1. If \( A \rightarrow \alpha \cdot \beta \) is in \( I_i \) and \( \text{GOTO}(I_i, \alpha) = I_j \) where \( \alpha \) is a terminal, set \( \text{ACTION}[I_i, \alpha] \) to “shift \( j \)”.
  2. If \( A \rightarrow \alpha \cdot \beta \) is in \( I_i \), set \( \text{ACTION}[I_i, \alpha] \) to “reduce \( A \rightarrow \alpha \cdot \beta \)” for all terminal \( \alpha \) in FOLLOW(A).
  3. If \( S' \rightarrow S \cdot \) is in \( I_i \), set \( \text{ACTION}[I_i, \$] = \text{accept} \).
  4. If \( \text{GOTO}(I_i, A) = I_j \), set \( \text{GOTO}[I_i, A] = I_j \).
  5. Set all other entries to “error”.
  6. Set initial state to be \( I_i \) with \( S' \rightarrow .S \).

Limitation of SLR(1) Parser

- Unfortunately, many unambiguous grammars are not SLR(1) grammars.

LR(1) Parsing

- Conflict arises because LR(0) states do not encode enough left context — the previous example, reduction \( R \rightarrow L \) is wrong upon input "because "R = . . .

- Solution: split LR(0) states by adding terminals to states, for example, \( [A = \rightarrow \alpha \cdot , \alpha] \) results in reduction only if next symbol is \( \alpha \).

Building LR(1) DFA

- Construct LR(1) NFA with all LR(1) items of \( G \) as states, connect states by moving the dot; then convert the NFA to DFA.

- Construct the LR(1) DFA directly (see the Dragon book).

- Given a set of LR(1) items \( I \), \( \text{CLOSURE}(I) \) is now defined as
  - repeat for each item \( [A \rightarrow \alpha \cdot \beta \cdot , \alpha] \) in \( I \) and each production \( B \rightarrow \gamma \) and each terminal \( b \) in \( \text{FIRST}(\beta \cdot \alpha) \)
    - add \( [B \rightarrow \gamma, b] \) to \( I \), if it’s not in \( I \) until \( I \) does not change.

Canonical LR(0) collection ---

| \( I_0 \) | \( S' \rightarrow .S \) |
| \( I_3 \) | \( S \rightarrow R \) |
| \( I_6 \) | \( S \rightarrow L \cdot R \) |
| \( I_4 \) | \( L \rightarrow \cdot \cdot .R \) |
| \( I_7 \) | \( L \rightarrow \cdot \cdot \cdot R \) |
| \( I_1 \) | \( S' \rightarrow S \) |
| \( I_5 \) | \( L \rightarrow \cdot \cdot \cdot \cdot R \) |
| \( I_8 \) | \( S \rightarrow L \) |

State 2 has a shift/reduce conflict on "=": shift 6 or reduce \( R \rightarrow L \).
Constructing LR(1) Parser

- **Canonical LR(1) collection** is computed by the following procedure:

  \[ I_0 = \text{CLOSURE}(\{ S' \to \cdot S, \cdot \}) \text{ and } C = \{ I_0 \} \]

  repeat

  for each \( I \in C \) and grammar symbol \( X \)

  \[ T = \text{GOTO}(I, X); \text{ if } T \neq \emptyset \text{ and } T \notin C \text{ then } C = C \cup \{ T \}; \]

  until \( C \) does not change

  Resulting LR(1) DFA: \( C \) is the set of states; \( \text{GOTO} \) is the transition table

- From the LR(1) DFA, we can construct the parsing table — LR(1) parsing table. The parser based on LR(1) parsing table is called LR(1) parser. The LR(1) grammars are those whose LR(1) parsing table does not contain any conflicts (no duplicate entries).

- **Example:**

  \[
  \begin{align*}
  S' & \rightarrow S \\
  S & \rightarrow C \ C \\
  C & \rightarrow c \ C \mid d
  \end{align*}
  \]

LALR(1) Parsing

- **Bad News:** LR(1) parsing tables are too big; for PASCAL, SLR tables has about 100 states, LR table has about 1000 states.

- **LALR (Lookahead-LR) parsing tables** have same number of states as SLR, but use lookahead for reductions. The LALR(1) DFA can be constructed from the LR(1) DFA.

- **LALR(1) states** can be constructed from LR(1) states by merging states with same core, or same LR(0) items, and union their lookahead sets.

  - **Merging** I8: \( C \rightarrow c \cdot, c/d \) into a new state I89: \( C \rightarrow c \cdot, c/d/\$
  
  - **Merging** I3: \( C \rightarrow c \cdot, c \mid c/d \) into a new state I36: \( C \rightarrow c \cdot, c/d/\$

Summary: LR Parser

- **Relation of three LR parsers:** \( LR(1) > LALR(1) > SLR(1) \)

- **Most programming language constructs are LALR(1).** The LR(1) is unnecessary in practice, but the SLR(1) is not enough.

- **YACC is an LALR(1) Parser Generator.**

- When parsing ambiguous grammars using LR parsers, the parsing table will contain multiple entries. We can specify the precedence and associativity for terminals and productions to resolve the conflicts. YACC uses this trick.

- **Other Issues** in parser implementation: 1. compact representation of parsing table 2. error recovery and diagnosis.
Top-Down Parsing

- Starting from the start symbol and “guessing” which production to use next step. It often uses next input token to guide “guessing”.

Example:

$$S \rightarrow c \ A \ d$$
$$A \rightarrow ab \mid a$$

input symbols: cad
we are looking ahead only one at a time!

decide to use 1st alternative of A

“c” matches
try to decide which rule of A to use here?

‘guessed wrong, backtrack and try another one!’

Top-Down Parsing (cont’d)

- Typical implementation is to write a recursive procedure for each non-terminal (according to the r.h.s. of each grammar rule)

Grammar:

$$E \rightarrow T\ E'$$
$$E' \rightarrow + \ T\ E'$$
$$T \rightarrow F\ T'$$
$$T' \rightarrow \ast \ T'$$
$$T' \rightarrow \epsilon$$
$$F \rightarrow \text{id} \mid (\ E)$$

Algorithm: Recursive Descent

- Parsing Algorithm (using 1-symbol lookahead in the input)

1. Given a set of grammar rules for a non-terminal A

   $$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

   we choose proper alternative by looking at first symbol it derives ---- the next input symbol decides which \( \alpha_i \) we use

2. for \( A \rightarrow \epsilon \), it is taken when none of the others are selected

- Algorithm: constructing a recursive descent parser for grammar G

1. transform grammer G to \( G' \) by removing left-recursions and do the left-factoring.

2. write a (recursive) procedure for each non-terminal in \( G' \)

Recursive Descent Parsing

- The previously referred top-down parsing method is often called recursive descent parsing!

- Main challenges:

  1. back-tracking is messy, difficult and inefficient
     (solution: use input “lookahead” to help make the right choice)

  2. more alternatives --- even if we use one lookahead input char, there are still \( > 1 \) rules to choose --- \( A \rightarrow ab \mid a \)
     (solution: rewrite the grammar by left-factoring)

  3. left-recursion might cause infinite loop
     what is the procedure for \( E \rightarrow E + E \) ?
     (solution: rewrite the grammar by eliminating left-recursions)

  4. error handling --- errors detected “far away” from actual source.
Left Recursion Elimination

- **Elimination of Left Recursion** (useful for top-down parsing only)

  replace productions of the form
  \[ A \rightarrow A \beta | \alpha \]
  with
  \[ A \rightarrow \beta A' \]
  \[ A' \rightarrow \alpha A' | \epsilon \]

  (yields different parse trees but same language)

  **example:**
  \[
  E \rightarrow E + T | T \\
  T \rightarrow T * F | F \\
  F \rightarrow ( E ) | id
  \]

  become
  \[
  E \rightarrow T E' \\
  E' \rightarrow + T E' | \epsilon \\
  T \rightarrow F T' \\
  T' \rightarrow * F T' | \epsilon \\
  F \rightarrow ( E ) | id
  \]

  **Important:** read Appel pp 51 - 53 for details

Left Factoring

- **Some grammars are unsuitable for recursive descent, even if there is no left recursion**

  "dangling-else"

  \[
  stmt \rightarrow if \ expr \ then \ stmt \\
  | if \ expr \ then \ stmt \ else \ stmt \\
  | ....
  \]

  input symbol if does not uniquely determine alternative.

- **Left Factoring** --- factor out the common prefixes (see AHU pp 178)

  change the production
  \[ A \rightarrow x y | x z \]
  to
  \[ A \rightarrow x A' \]
  \[ A' \rightarrow y | z \]

  thus
  \[ stmt \rightarrow if \ expr \ then \ stmt \ S' \]
  \[ S' \rightarrow else \ stmt | \epsilon \]

Predictive Parsing

- **Predictive parsing** is just table-driven recursive descent; it contains:

  A **parsing stack** --- contains terminals and non-terminals
  A **parsing table** : a 2-dimensional table \( M[X, a] \) where \( X \) is non-terminal, \( a \) is terminal, and table entries are grammar productions or error indicators.

  **algorithm**
  \( S \) is end-of-file, \( S \) is start symbol

  push(\$); push(S);
  while top \( \neq \$ \) do {
    a <- the input char
    if top is a terminal or \( \$ \) then
      if top \( = \) a then
        pop(); advance()
      else err()
    else if M(top, a) is \( X \rightarrow Y_1 Y_2 \ldots Y_n \) then
      (pop(),
      push(Y_1), \ldots, push(Y_n))
    else err()}

  rest of \( M \) is error

  **FIRST**(\( a \)) is a set of terminals (plus \( \epsilon \)) that begin strings derived from \( a \), where \( a \) is any string of non-terminals and terminals.

  **FOLLOW**(\( A \)) is a set of terminals that can follow \( A \) in a sentential form, where \( A \) is any non-terminal

Constructing Predictive Parser

- **The key is to build the parse table** \( M[A, a] \)

  for each production \( A \rightarrow \alpha \) do
  for each \( a \in \text{FIRST}(\alpha) \) do
    add \( A \rightarrow \alpha \) to \( M[A, a] \)
  if \( \epsilon \in \text{FIRST}(\alpha) \) then
    for each \( b \in \text{FOLLOW}(A) \) do
      add \( A \rightarrow \alpha \) to \( M[A, b] \)
  rest of \( M \) is error

  **FIRST**(\( a \)) is a set of terminals (plus \( \epsilon \)) that begin strings derived from \( a \), where \( a \) is any string of non-terminals and terminals.

  **FOLLOW**(\( A \)) is a set of terminals that can follow \( A \) in a sentential form, where \( A \) is any non-terminal.
First & Follow

• To compute $\text{FIRST}(X)$ for any grammar symbol $X$:

$\text{FIRST}(X) = \{X\}$, if $X$ is a terminal;
$\text{FIRST}(X) = \text{FIRST}(X) \cup \{a\}$, if $X \rightarrow a a$;
$\text{FIRST}(X) = \text{FIRST}(X) \cup \{\varepsilon\}$, if $X \rightarrow \varepsilon$; and
$\text{FIRST}(X) = \text{FIRST}(X) \cup \text{FIRST}(Y_1 Y_2 \ldots Y_k)$, if $X \rightarrow Y_1 Y_2 \ldots Y_k$.

repeat until nothing new is added to any $\text{FIRST}$

• $\text{FIRST}(Y_1 Y_2 \ldots Y_k) = \text{FIRST}(Y_1) - \{\varepsilon\}$

$\cup \text{FIRST}(Y_2) - \{\varepsilon\}$ if $\varepsilon \in \text{FIRST}(Y_2)$

$\cup \text{FIRST}(Y_3) - \{\varepsilon\}$ if $\varepsilon \in \text{FIRST}(Y_1 Y_2)$

......................

$\cup \text{FIRST}(Y_k) - \{\varepsilon\}$ if $\varepsilon \in \text{FIRST}(Y_1 \ldots Y_{k-1})$

$\cup \{\varepsilon\}$ if all $\text{FIRST}(Y_i)_{i=1 \ldots k}$ contain $\varepsilon$.

First & Follow (cont’d)

• To compute $\text{FOLLOW}(X)$ for any non-terminal $X$:

$\text{FOLLOW}(S) = \text{FOLLOW}(S) \cup \{\$\}$, if $S$ is start symbol;
$\text{FOLLOW}(B) = \text{FOLLOW}(B) \cup (\text{FIRST}(\beta) - \{\varepsilon\})$, if $A \rightarrow \alpha B \beta$ and $\beta \neq \varepsilon$

$\text{FOLLOW}(B) = \text{FOLLOW}(B) \cup \text{FOLLOW}(A)$ if $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ and $\varepsilon \in \text{FIRST}(\beta)$

• Example:

$\text{FOLLOW}(E) = \{\$\}$, reason: $E$ is start symbol
$\text{FOLLOW}(E') = \text{FOLLOW}(E)$
$\text{FOLLOW}(E') = \{\$\}$, reason: $E \rightarrow T E'$

(Read Appel pp 47 - 53 for detailed examples)

Summary: LL(1) Grammars

• A grammar is $\text{LL}(1)$ if parsing table $M[A, a]$ has no duplicate entries, which is equivalent to specifying that for each production

$A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$

1. All $\text{FIRST}(\alpha_i)$ are disjoint.

2. At most one $\alpha_i$ can derive $\varepsilon$; in that case, $\text{FOLLOW}(A)$ must be disjoint from $\text{FIRST}(\alpha_1) \cup \text{FIRST}(\alpha_2) \cup \ldots \cup \text{FIRST}(\alpha_n)$

• Left-recursion and ambiguity grammar lead to multiple entries in the parsing table. (Try the dangling-else example)

• The main difficulty in using (top-down) predicative parsing is in rewriting a grammar into an $\text{LL}(1)$ grammar. There is no general rule on how to resolve multiple entries in the parsing table.