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
  \[(s_0X_1s_1X_2s_2\ldots XMs_m, a_1a_2a_3\ldots a_n\$$)\]
  next “move” is determined by $s_m$ and $a_i$
- **Parsing tables**: ACTION[s,a] and GOTO[s,X]

  **Table A** ACTION[s,a] --- $s$: state, a: terminal
  - its entries
    1. (1) shift $s_k$
    2. (2) reduce $A \rightarrow \beta$
    3. (3) accept
    4. (4) error
  **Table G** GOTO[s,X] --- $s$: state, X: non-terminal
  - its entries are states

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 $G' = (T,N \cup \{ S' \}, P \cup \{ S' \rightarrow S \}, S')$ --- 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 $A \rightarrow .XYZ$ and $A \rightarrow .X.YZ$
  - For $A \rightarrow \varepsilon$, its items are just $A \rightarrow \varepsilon$
LR(0) items and LR(0) DFA

- Informally, item \( A \rightarrow X \cdot YZ \) 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:

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

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:

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

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.
- Convert NFA to DFA using subset construction algorithm.
- The states of the resulting LR(0) DFA --- \( C = \{ I_1, I_2, \ldots, I_n \} \) 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.
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 \(A \rightarrow \alpha.a\beta\) is in \(I_i\) and \(\text{GOTO}(I_i, a) = I_j\) where \(a\) is a terminal, set \(\text{ACTION}[I_i, a]\) to “shift \(j\)”.
  2. If \(A \rightarrow \alpha.\) is in \(I_i\), set \(\text{ACTION}[I_i, \alpha]\) to “reduce \(A \rightarrow \alpha\)” for all terminal \(\alpha\) in FOLLOW(\(A\)).
  3. If \(S' \rightarrow S\). is in \(I_i\), set \(\text{ACTION}[I_i, \$]\) to “accept”
  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

S -> L = R R | R L
L -> *R id | R
R -> .L | id

Canonical LR(0) collection ——

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I0</td>
<td>I3</td>
<td>I4</td>
<td>I5</td>
<td>I6</td>
<td>I7</td>
<td>I8</td>
</tr>
</tbody>
</table>

LR(1) Parsing

• Conflict arises because LR(0) states do not encode enough left context — in the previous example, reduction \(R \rightarrow L\) is wrong upon input \(=\) because “\(\ldots\) never appears in right-sentential form.”

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

• An LR(1) term is in the form of \([A \rightarrow \alpha.\beta, a]\) where \(A \rightarrow \alpha\beta\) is a production and \(a\) is a terminal or \$. 

• To build LR(1) parsing table —— we first build LR(1) DFA —— then construct the parsing table using the same SLR(1) algorithm except

  2. only if \([A \rightarrow \alpha, a]\) is in \(I_i\), then set \(\text{ACTION}[I_i, a]\) to “reduce \(A \rightarrow \alpha\)”

• To way to build LR(1) DFA —— from NFA -> DFA or build DFA directly.

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.

  1. for each item \([A \rightarrow \alpha.aX\beta, a]\)

  2. for each pair \([A \rightarrow \alpha.a\beta, a]\) \(B \rightarrow \gamma\) and \(b\) in FIRST(\(\beta\))

• 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 \([A \rightarrow \alpha.a\beta, a]\) in I and each production \(B \rightarrow \gamma\) and each terminal \(b\) in FIRST(\(\beta\))

    add \([B \rightarrow \gamma, a]\) to I, if it's not in I

    until I does not change
Constructing LR(1) Parser

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

  \[ I_0 = \text{CLOSURE}([S' \rightarrow .S, \$]) \quad \text{and} \quad C = \{I_0\} \]

  repeat

  
  \[ \text{for each } I \in C \text{ and grammar symbol } X \]

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

  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:**

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

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 \( I_8 \) : \( C \rightarrow cCC.c/d \), \( c/\mid d \) \( \rightarrow \) \( I_9 \) : \( C \rightarrow cCC.\$ \)

  \text{into a new state} \( I_{89} : C \rightarrow cCC.c/d.\$ \)

  Merging \( I_3 \) : \( C \rightarrow c.C.c/\mid d \) \( \rightarrow \) \( I_6 \) : \( C \rightarrow c.C.\$ \)

  \( C \rightarrow .c.C.c/\mid d \) \( \rightarrow \$ \)

  \( C \rightarrow .d.C.c/\mid d \) \( \rightarrow \$ \)

  \text{into a new state} \( I_{36} : C \rightarrow c.C.c/d.\$ \)

  \( C \rightarrow .c.C.c/d.\$ \)

  \( C \rightarrow .d.C.c/d.\$ \)

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:

\[
S \rightarrow c \ A \ d \\
A \rightarrow ab \ | \ a
\]

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' \ | \ \epsilon \\
T \rightarrow F T' \\
T' \rightarrow * F T' \ | \ \epsilon \\
F \rightarrow \text{id} \ | \ (E)
```

```
fun e() = (t(); eprime())
and eprime() =
  if (c = "+")
    then (advance(); t(); eprime())
  and t() = (f(); tprime())
and tprime() =
  if (c = "*")
    then (advance(); f(); tprime())
and f() =
  (if (c = "id") then advance()
   else if (c = "(") then
   advance(); e();
   if (c="") then advance() else err()
   else err)
```

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 --- (solution: rewrite the grammar by left-factoring)
  3. left-recursion might cause infinite loop (solution: rewrite the grammar by eliminating left-recursions)
  4. error handling --- errors detected “far away” from actual source.

Algorithm: Recursive Descent

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

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

\[
A \rightarrow \alpha_1 \ | \ \alpha_2 \ | \ldots \ | \ \alpha_n
\]

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 \(G'\)
**Left Recursion Elimination**

- **Elimination of Left Recursion** (useful for top-down parsing only)
  
  replace productions of the form \( A \rightarrow A \alpha | \beta \)
  
  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

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

  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:**

  \[
  S \ is \ end-of-file, \ S \ is \ start \ symbol \\
  push(S); \ push(S); \\
  while top <> $ do \\
  \quad a <- \ the \ input \ char \\
  \quad if top is a terminal or $ then \\
  \quad \quad if top == a then \\
  \quad \quad \quad \quad \quad pop(); \ advance() \\
  \quad \quad else err() \\
  \quad else if M[top, a] is X -> Y_1 Y_2 ... Y_n then \\
  \quad \quad (pop(); \\
  \quad \quad \quad \quad \quad push(Y_1); ...; push(Y_n); \\
  \quad \quad else err() \\
  \]

**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 \( A \rightarrow a \) 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

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

- \( \text{FOLLOW}(A) \) 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 \( X \):

\[
\text{FIRST}(X) = \begin{cases} 
X, & \text{if } X \text{ is a terminal;} \\
\text{FIRST}(X) \cup \{a\}, & \text{if } X \rightarrow a \text{ ; and} \\
\text{FIRST}(X) \cup \text{FIRST}(Y_1Y_2...Y_k), & \text{if } X \rightarrow Y_1Y_2...Y_k \text{.} 
\end{cases}
\]

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

- \( \text{FIRST}(Y_1Y_2...Y_k) = \text{FIRST}(Y_1) \cup \text{FIRST}(Y_2) \cup \ldots \cup \text{FIRST}(Y_k) \)

\( \text{FIRST}(X) = \text{FIRST}(X) \cup \text{FIRST}(Y_1Y_2...Y_k), \) 
\( \text{if } X \rightarrow Y_1Y_2...Y_k \text{.} \)

First & Follow (cont’d)

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

\[
\text{FOLLOW}(S) = \text{FOLLOW}(S) \cup \{\$\}, \quad \text{if } S \text{ is start symbol;}
\]

\[
\text{FOLLOW}(A) = \text{FOLLOW}(A) \cup (\text{FIRST}(\beta) - \{\epsilon\}), \quad \text{if } A \rightarrow \alpha \beta \text{ and } \beta \neq \epsilon
\]

\[
\text{FOLLOW}(B) = \text{FOLLOW}(B) \cup \text{FOLLOW}(A) \\
\text{if } A \rightarrow \alpha B \text{ or } A \rightarrow \alpha B \beta \text{ and } \epsilon \in \text{FIRST}(B)
\]

- Example:

\[
\text{FOLLOW}(E) = \{\$, \text{ reason } E \text{ is start symbol}
\]
\[
\text{FOLLOW}(E') = \text{FOLLOW}(E) - \{\$\}, \quad E' \rightarrow T
\]

(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 \( \epsilon \); 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.