Revisit the example

$\epsilon$-closure(0) = {0, 1, 2, 4, 7} = A
Trans(A, a) = {1, 2, 3, 4, 6, 7, 8} = B
Trans(A, b) = {1, 2, 4, 5, 6, 7} = C
Trans(B, a) = {1, 2, 3, 4, 6, 7, 8} = B
Trans(B, b) = {1, 2, 4, 5, 6, 7, 9} = D
Trans(C, a) = {1, 2, 3, 4, 6, 7, 8} = B
Trans(C, b) = {1, 2, 4, 5, 6, 7} = C
Trans(D, a) = {1, 2, 3, 4, 6, 7, 8} = B
Trans(D, b) = {1, 2, 4, 5, 6, 7, 10} = E (mark as end state because 10 is included)
Trans(E, a) = {1, 2, 3, 4, 6, 7, 8} = B
Trans(E, b) = {1, 2, 4, 5, 6, 7} = C

Transformed DFA
Minimizing the DFA

• Insight
  – Identify equivalent state sets if all states have the same transitions
  – Merge equivalent states as a new state in the refined DFA

Minimizing the DFA

Initially partition the state set into two groups:
(1) Group I: the states with only final states, and
(2) Group II: the states with only non-final states

for each group $G$ do
  partition $G$ into subgroups such that for any states $s$ and $t$ in $G$, for all input symbols, they have transitions to states in the same group
  replace $G$ with identified subgroups
Revisit the example

Initially, two partitions:
\( G_1 = \{E\}, \ G_2 = \{A, B, C, D\} \)

\( G_1 \) cannot be further partitioned

\[
\begin{array}{c|cc}
\text{State} & \text{Input symbol} \\
\hline
 & a & b \\
A & B & C \\
B & B & D \\
C & B & C \\
D & B & E \\
E & B & C \\
\end{array}
\]

Trans \( (G_2, a) = \{B\} \subseteq G_2 \)

Trans \( (G_2, b) = \{C, D, E\} \), the resulting set does not belong to the same group

Partition \( G_2 \) into \( \{A, B, C\} = G_3, \{D\} = G_4 \)

Trans \( (G_3, a) = \{B\} \subseteq G_3 \)

Trans \( (G_3, b) = \{C, D\} \), the resulting set does not belong to the same group

Partition \( G_3 \) into \( \{A, C\} = G_5, \{B\} = G_6 \)

Trans \( (G_5, a) = \{B\} = G_6 \)

Trans \( (G_5, b) = \{C\} \subseteq G_5 \)

Therefore, the resulting partition is:
\( \{A, C\}, \{B\}, \{D\}, \{E\} \)

Refined DFA

![Refined DFA Diagram]
Constructing the Lexical Analysis

- Convenient utility subprograms:
  - `getChar` - gets the next character of input, puts it in `nextChar`, determines its class and puts the class in `charClass`
  - `addChar` - puts the character from `nextChar` into the place the token is being accumulated: `nextToken`
Implementation Pseudo-code

static TOKEN nextToken;
static CHAR_CLASS charClass;
void lex() {
  getChar();
  switch (charClass) {
    case LETTER:
      addChar();
      getChar();
      while (charClass == LETTER || charClass == DIGIT)
      {
        addChar();
        getChar();
      }
      return; //nextToken = ID
    case DIGIT:
      addChar();
      getChar();
      while (charClass == DIGIT) {
        addChar();
        getChar();
      }
      return; //nextToken = INT_LIT
    default: report error();
  }  /* End of switch */
}  /* End of function lex */

Implementation (Cont’d)

...
Key Points about Scanner

• Nearly universal rule
  – Always take the longest possible token from the input
    • foobar is never parsed to foo or fooba
• Regular expressions “generate” a regular language, while DFAs “recognize” it

Parser

• By analogy to RE and DFAs, a context-free grammar (CFG) is a generator for a context-free language, while a parser is a language recognizer
• Responsibilities
  – Generate a parse tree, report syntax errors if any
Two Classes of Grammars

• Left-to-right, Leftmost derivation (LL)
• Left-to-right, Rightmost derivation (LR)
• We can build parsers for these grammars that run in linear time

<table>
<thead>
<tr>
<th>Grammar Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LL</strong></td>
</tr>
<tr>
<td>E → T E'</td>
</tr>
<tr>
<td>E' → + T E'</td>
</tr>
<tr>
<td>T → F T'</td>
</tr>
<tr>
<td>T' → * F T'</td>
</tr>
</tbody>
</table>
Two Categories of Parsers

• LL(1) Parsers
  – L: scanning the input from left to right
  – L: producing a leftmost derivation
  – 1: using one input symbol of lookahead at each step to make parsing action decisions

• LR(1) Parsers
  – L: scanning the input from left to right
  – R: producing a rightmost derivation in reverse
  – 1: the same as above

Two Categories of Parsers

• LL(1) parsers (predicative parsers)
  – Top down
    • Build the parse tree from the root
    • Find a left most derivation for an input string

• LR(1) parsers (shift-reduce parsers)
  – Bottom up
    • Build the parse tree from leaves
    • Reducing a string to the start symbol of a grammar
Top-down parsing algorithms

• Recursive predictive parsing
  – Recursive-descent parsing
• Non-recursive predictive parsing
  – LL(1): Table-driven parsing

Motivating Example

• Consider the grammar
  \[ S \rightarrow cAd \]
  \[ A \rightarrow ab \mid a \]
• Input string: \( w = cad \)
• How to build a parse tree top-down?
Recursive-Descent Parsing

- Initially create a tree containing a single node \( S \) (the start symbol)
- Apply the \( S \)-rule to see whether the first token matches
  - If matches, expand the tree
    - Apply the \( A \)-rule to the leftmost nonterminal \( A \)
      - Since the first token matches both alternatives (\( A_1 \) and \( A_2 \)), randomly pick one (e.g., \( A_1 \)) to apply

Recursive-Descent Parsing

- Since the third token \( d \) does not match \( b \), report failure and go back to \( A \) to try another alternative
- Rollback to the state before applying \( A_1 \) rule, and then apply the alternative rule
- The third token matches, so parsing is successfully done
Recursive-Descent Parsing Algorithm

Suppose we have a scanner which generates the next token as needed. Given a string, the parsing process starts with the start symbol rule:

if there is only one RHS then
  for each terminal in the RHS
    compare it with the next input token
    if they match, then continue
    else report an error
  for each nonterminal in the RHS
    call its corresponding subprogram and try match
    if no match is found, then report an error
else // there is more than one RHS
  choose the RHS based on the next input token (the lookahead)
  for each chosen RHS
    call the corresponding subprogram and try match
    if no match is found, then report an error

Constructing Parser

• Utility program
  – match(...) sees what it expects to see, and do corresponding processing
An Example (one RHS)

/* Function expr Parses strings in the language generated by the rule:
<expr> -> <term> {(+ | -) <term>} */

void expr() {
   /* Parse the first term */
   term();

   /* As long as the next token is + or -, call lex to get the next token, and parse the next term */
   while (nextToken == PLUS_CODE ||
      nextToken == MINUS_CODE) {
      lex();
      term();
   }
}

Another Example (multiple RHS)

/* Function factor Parses strings in the language generated by the rule:
<factor> -> id  |  (<expr>) */

void factor() {
   if (nextToken == ID_CODE) {
      lex();
   }
   else if (nextToken == LEFT_PAREN_CODE) {
      lex();
      expr();
      if (nextToken == RIGHT_PAREN_CODE) {
         lex();
      }
      else
         error();
   }
   else error(); /* Neither RHS matches */
}
Key points about recursive-descent parsing

• Recursive-descent parsing may require backtracking
• LL(1) does not allow backtracking
  – By only looking at the next input token, we can always precisely decide which rule to apply
• By carefully designing a grammar, i.e., LL(1) grammar, we can avoid backtracking

Two Obstacles to LL(1)-ness

• Left recursion
  – E.g., id_list -> id_list_prefix ;
    id_list_prefix -> id_list_prefix, id | id
  – When the next token is id, which rule should we apply?
• Common prefixes
  – E.g., A -> ab | a
  – When the next token is a, which rule should we apply?
**LL(1) Grammar**

- Grammar which can be processed with LL(1) parser
- Non-LL grammar can be converted to LL(1) grammar via:
  - Left-recursion elimination
  - Left factoring by extracting common prefixes

**Left-Recursion Elimination**

- Replace left-recursion with right-recursion
  
  id_list -> id_list_prefix ;
  id_list_prefix -> id_list_prefix, id | id
  =>
  id_list -> id id_list_tail
  id_list_tail -> ; | , id id_list_tail
Left Factoring

• Extract the common prefixes, and introduce new nonterminals as needed
  \[ A \rightarrow ab \mid a \]
  \[ \Rightarrow \]
  \[ A \rightarrow aB \]
  \[ B \rightarrow b \mid \varepsilon \]

Non-LL Languages

• Simply eliminating left recursion and common prefixes is not guaranteed to make LL(1)
  
• An example in Pascal:
    \[ stmt \rightarrow \text{if condition then\_clause else\_clause} \]
    \[ \quad \mid \text{other\_stmt} \]
    \[ \text{then\_clause} \rightarrow \text{then stmt} \]
    \[ \text{else\_clause} \rightarrow \text{else stmt} \mid \varepsilon \]

• How to parse “if C1 then if C2 then S1 else S2” ?
Non-LL Languages

• Define “disambiguating rule”, use it together with ambiguous grammar to parse top-down
  – E.g., in the case of a conflict between two possible productions, the one to use is the one that occurs first, textually in the grammar
  – to pair the else with the nearest then

• “Disambiguating rule” can be also defined for bottom-up parsing