Concepts Introduced in Chapter 4

- Grammars
  - Context-Free Grammars
  - Derivations and Parse Trees
  - Ambiguity, Precedence, and Associativity
- Top Down Parsing
  - Recursive Descent, LL
- Bottom Up Parsing
  - SLR, LR, LALR
- Yacc
- Error Handling

Example of a Grammar

\[
\begin{align*}
\text{expression} & \rightarrow \text{expression} + \text{term} \\
\text{expression} & \rightarrow \text{expression} - \text{term} \\
\text{expression} & \rightarrow \text{term} \\
\text{term} & \rightarrow \text{term} \ast \text{factor} \\
\text{term} & \rightarrow \text{term} / \text{factor} \\
\text{term} & \rightarrow \text{factor} \\
\text{factor} & \rightarrow ( \text{expression} ) \\
\text{factor} & \rightarrow \text{id}
\end{align*}
\]

Grammars

\[G = (N, T, P, S)\]

1. \(N\) is a finite set of nonterminal symbols
2. \(T\) is a finite set of terminal symbols
3. \(P\) is a finite subset of \((N \cup T)^* N (N \cup T)^* \times (N \cup T)^*\)

An element \((\alpha, \beta) \in P\) is written as \(\alpha \rightarrow \beta\)

and is called a production.

4. \(S\) is a distinguished symbol in \(N\) and is called the start symbol.

Advantages of Using Grammars

- Provides a precise, syntactic specification of a programming language.
- For some classes of grammars, tools exist that can automatically construct an efficient parser.
- These tools can also detect syntactic ambiguities and other problems automatically.
- A compiler based on a grammatical description of a language is more easily maintained and updated.
Role of a Parser in a Compiler

- Detects and reports any syntax errors.
- Produces a parse tree from which intermediate code can be generated.

Conventions for Specifying Grammars in the Text

- terminals
  - lower case letters early in the alphabet (a, b, c)
  - punctuation and operator symbols [ ( ), , ', +, −]
  - digits
  - boldface words (if, then)
- nonterminals
  - uppercase letters early in the alphabet (A, B, C)
  - S is the start symbol
  - lower case words

Conventions for Specifying Grammars in the Text (cont.)

- grammar symbols (nonterminals or terminals)
  - upper case letters late in the alphabet (X, Y, Z)
- strings of terminals
  - lower case letters late in the alphabet (u, v, ..., z)
- sentential form (string of grammar symbols)
  - lower case Greek letters (α, β, γ)

Chomsky Hierarchy

A grammar is said to be
1. **regular** if it is
   where each production in P has the form
   a. **right-linear**
      \[ A \rightarrow wB \text{ or } A \rightarrow w \]
   b. **left-linear**
      \[ A \rightarrow Bw \text{ or } A \rightarrow w \]
   where \( A, B \in N \) and \( w \in T^* \)
2. context-free: each production in $P$ is of the form $A \rightarrow L$ where $A \in N$ and $L = (N \cup T)^*$

3. context-sensitive: each production in $P$ is of the form $\alpha A \beta \rightarrow \alpha L \beta$ where $|\alpha| \leq |\beta|$.

4. unrestricted: if each production in $P$ is of the form $\alpha A \beta \rightarrow \alpha L \beta$ where $A \in N$, $\alpha$, $\beta \in (N \cup T)^*$.

**Derivation (cont.)**

Parse Tree

Given a context-free grammar, a Parse Tree has the properties:

- The root is labeled by the start symbol.
- Each leaf is labeled by a token or $\epsilon$.
- Each interior node is labeled by a nonterminal.
- If $A$ is a nonterminal labeling some interior node and $X_1, X_2, X_3, \ldots, X_n$ are the labels of the children of that node from left to right, then $A \rightarrow L_{x_1} L_{x_2} L_{x_3} \cdots L_{x_n}$ is a production of the grammar.

**Parse Tree**

$$ ( p + p ) \rightarrow \epsilon $$

Thus $E$ derives $p + p$.

**Derivation**

4. unrestricted: each production in $P$ is of the form $\alpha \rightarrow \beta$ where $|\beta| \leq |\alpha|$

3. context-sensitive: each production in $P$ is of the form $\alpha \rightarrow \beta$ where $A \in N$ and $\alpha, \beta \in (N \cup T)^*$.

2. context-free: each production in $P$ is of the form $A \rightarrow L$.

**Chomsky Hierarchy (cont.)**
**Example of a Parse Tree**

```
list → list + digit | list - digit | digit
```

```
Example of an Ambiguous Grammar
```

\[
\text{string} \rightarrow \text{string} + \text{string} \\
\text{string} \rightarrow \text{string} - \text{string} \\
\text{string} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

**Parse Tree (cont.)**

- **Yield**
  - the leaves of the parse tree read from left to right, or
  - the string derived from the nonterminal at the root of the parse tree

- An ambiguous grammar is one that can generate two or more parse trees that yield the same string.

**Precedence**

By convention

\[
9 + 5 * 2
\]

* has higher precedence than + because it takes its operands before +

```
expr → expr + term | term
```

```
term → term * digit | digit
```

```
equiv
```

```
equiv
```

```
equiv
```

```
equiv
```

```
equiv
```

a. \[
\text{string} \rightarrow \text{string} + \text{string} \rightarrow \text{string} - \text{string} + \text{string} \\
\rightarrow 9 - \text{string} + \text{string} \rightarrow 9 - 5 + \text{string} \rightarrow 9 - 5 + 2
\]

b. \[
\text{string} \rightarrow \text{string} + \text{string} \rightarrow 9 - \text{string} \\
\rightarrow 9 - \text{string} + \text{string} \rightarrow 9 - 5 + \text{string} \rightarrow 9 - 5 + 2
\]
**Precedence (cont.)**

- If different operators have the same precedence then they are defined as alternative productions of the same nonterminal.

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} + \text{term} \mid \text{expr} - \text{term} \mid \text{term} \\
\text{term} & \rightarrow \text{term} \ast \text{factor} \mid \text{term} / \text{factor} \mid \text{factor} \\
\text{factor} & \rightarrow \text{digit} \mid (\text{expr})
\end{align*}
\]

**Associativity**

By convention

\[
9 - 5 - 2 \quad \text{left (operand with - on both sides is taken by the operator to its left)}
\]

\[
a = b = c \quad \text{right}
\]

**Eliminating Ambiguity**

- Sometimes ambiguity can be eliminated by rewriting a grammar.

\[
\text{stmt} \rightarrow \text{if expr then stmt} \\
\mid \text{if expr then stmt else stmt} \\
\mid \text{other}
\]

- How do we parse:

\[
\text{if E1 then if E2 then S1 else S2}
\]

**Eliminating Ambiguity (cont.)**

- \[
\text{stmt} \rightarrow \text{matched_stmt} \\
\mid \text{unmatched_stmt}
\]

- \[
\text{matched_stmt} \rightarrow \text{if expr then matched_stmt else matched_stmt} \\
\mid \text{other}
\]

- \[
\text{unmatched_stmt} \rightarrow \text{if expr then stmt} \\
\mid \text{if expr then matched_stmt else unmatched_stmt}
\]
### Parsing

- Universal
- Top-down
  - recursive descent
  - LL
- Bottom-up
  - LR
    - SLR
    - canonical LR
    - LALR

### Top-Down vs Bottom-Up Parsing

- **top-down**
  - Have to eliminate left recursion in the grammar.
  - Have to left factor the grammar.
  - Resulting grammars are harder to read and understand.
- **bottom-up**
  - Difficult to implement by hand, so a tool is needed.

### Top-Down Parsing

Starts at the root and proceeds towards the leaves.

Recursive-Descent Parsing - a recursive procedure is associated with each nonterminal in the grammar.

#### Example

- type $\rightarrow$ simple $\mid$ id $\mid$ array [ simple ] of type
- simple $\rightarrow$ integer $\mid$ char $\mid$ num dotdot num

### Example of Recursive Descent Parsing

```c
void type() {
    if (lookahead == INTEGER || lookahead == CHAR || lookahead == NUM)
        simple();
    else if (lookahead == '^') {
        match('^'); match(ID);
    }
    else if (lookahead == ARRAY) {
        match(ARRAY); match('['); simple(); match(']');
        match(OF); type();
    } else {
        error();
    }
}
```

Followed by Fig. 4.12
Example of Recursive Descent Parsing (cont.)

```c
void simple() {
    void match(token t)
    if (lookahead == INTEGER)
        match(INTEGER);
    else if (lookahead == CHAR)
        match(CHAR);
    else if (lookahead == NUM) {
        match(NUM);
        match(DOTDOT);
    } else
        error();
}
```

Top-Down Parsing (cont.)

- Predictive parsing needs to know what first symbols can be generated by the right side of a production.
- FIRST(\(\alpha\)) - the set of tokens that appear as the first symbols of one or more strings generated from \(\alpha\). If \(\alpha\) is \(\epsilon\) or can generate \(\epsilon\), then \(\epsilon\) is also in FIRST(\(\alpha\)).
- Given a production
  \[ A \rightarrow \alpha | \beta \]
  predictive parsing requires FIRST(\(\alpha\)) and FIRST(\(\beta\)) to be disjoint.

Eliminating Left Recursion

- Recursive descent parsing loops forever on left recursion.
- Immediate Left Recursion
  - Replace \( A \rightarrow A\alpha | \beta \) with \( A \rightarrow \beta A' \)
    - \( A' \rightarrow \alpha A' | \epsilon \)

Example:

<table>
<thead>
<tr>
<th>( A )</th>
<th>( \alpha )</th>
<th>( \beta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>+T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>*F</td>
</tr>
<tr>
<td>F</td>
<td>(E)</td>
<td>id</td>
</tr>
<tr>
<td>E'</td>
<td>+TE'</td>
<td>( \epsilon )</td>
</tr>
<tr>
<td>T</td>
<td>FT'</td>
<td></td>
</tr>
</tbody>
</table>

Eliminating Left Recursion (cont.)

In general, to eliminate left recursion given \( A_1, A_2, \ldots, A_n \)
for \( i = 1 \) to \( n \) do {
  for \( j = 1 \) to \( i-1 \) do {
    replace each \( A_i \rightarrow A_j \gamma \) with \( A_i \rightarrow \delta_1 \gamma | \ldots | \delta_k \gamma \)
    where \( A_j \rightarrow \delta_1 \gamma | \delta_2 \gamma | \ldots | \delta_k \gamma \) are the current \( A_j \) productions
  }
  eliminate immediate left recursion in \( A_i \) productions
  eliminate \( \epsilon \) transitions in the \( A_i \) productions
}

This fails only if cycles (\( A \rightarrow A \)) or \( A \rightarrow \epsilon \) for some \( A \).
Example of Eliminating Left Recursion

1. \( X \rightarrow YZ | a \)
2. \( Y \rightarrow ZX | Xb \)
3. \( Z \rightarrow XY | ZZ | a \)

\[ A1 = X \quad A2 = Y \quad A3 = Z \]

\( i = 1 \) (eliminate immediate left recursion)
nothing to do

Example of Eliminating Left Recursion (cont.)

\( i = 2, j = 1 \)
\[ Y \rightarrow Xb \Rightarrow Y \rightarrow ZX | YZb | ab \]
now eliminate immediate left recursion
\[ Y \rightarrow ZXY' | ab \quad Y' \rightarrow ZbY' | \epsilon \]
now eliminate \( \epsilon \) transitions
\[ Y \rightarrow ZXY' | abY' | ZX | ab \]
\[ Y' \rightarrow ZbY' | Zb \]

\( i = 3, j = 1 \)
\[ Z \rightarrow XY \Rightarrow Z \rightarrow YZY | aY | ZZ | a \]

Left-Factoring

\[ A \rightarrow \alpha \beta | \alpha \gamma \quad \Rightarrow \quad A \rightarrow \alpha A' \]
\[ A' \rightarrow \beta | \gamma \]

Example:

Linear factor
\[ \text{stmt} \rightarrow \text{if cond then stmt else stmt} \]
| \[ \text{if cond then stmt} \]
becomes
\[ \text{stmt} \rightarrow \text{if cond then stmt E} \]
\[ E \rightarrow \text{else stmt} | \epsilon \]

Useful for predictive parsing since we will know which production to choose.
Nonrecursive Predictive Parsing

- Instead of recursive descent, it is table-driven and uses an explicit stack. It uses
  1. a stack of grammar symbols ($ on bottom)
  2. a string of input tokens ($ on end)
  3. a parsing table [NT, T] of productions

Algorithm for Nonrecursive Predictive Parsing

1. If top == input == $ then accept
2. If top == input then
   pop top off the stack
   advance to next input symbol
   goto 1
3. If top is nonterminal
   fetch M[top, input]
   If a production
     replace top with rhs of production
   Else
     parse fails
     goto 1
4. Parse fails

First

\[ \text{FIRST}(\alpha) = \text{the set of terminals that begin strings derived from } \alpha. \text{ If } \alpha \text{ is } \varepsilon \text{ or generates } \varepsilon, \text{ then } \varepsilon \text{ is also in FIRST}(\alpha). \]

1. If X is a terminal then FIRST(X) = \{X\}
2. If X \rightarrow a\alpha, add a to FIRST(X)
3. If X \rightarrow \varepsilon, add \varepsilon to FIRST(X)
4. If X \rightarrow Y_1, Y_2, ..., Y_k and Y_1, Y_2, ..., Y_{i-1} *\Rightarrow \varepsilon
   where i \leq k
   Add every non \varepsilon in FIRST(Y_i) to FIRST(X)
   If Y_1, Y_2, ..., Y_k *\Rightarrow \varepsilon, add \varepsilon to FIRST(X)

FOLLOW

\[ \text{FOLLOW}(A) = \text{the set of terminals that can immediately follow } A \text{ in a sentential form.} \]

1. If S is the start symbol, add $ to FOLLOW(S)
2. If A \rightarrow a\beta, add FIRST(\beta) - \{\varepsilon\} to FOLLOW(B)
3. If A \rightarrow a\beta or A \rightarrow a\beta and \beta*\Rightarrow \varepsilon,
   add FOLLOW(A) to FOLLOW(B)
Example of Calculating FIRST and FOLLOW

<table>
<thead>
<tr>
<th>Production</th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → TE'</td>
<td>{ (, id }</td>
<td>{ ), $ }</td>
</tr>
<tr>
<td>E' → +TE'</td>
<td>{ +, ε }</td>
<td>{ ), $ }</td>
</tr>
<tr>
<td>T → FT'</td>
<td>{ (, id }</td>
<td>{ +, ), $ }</td>
</tr>
<tr>
<td>T' → *FT'</td>
<td>{ *, ε }</td>
<td>{ +, ), $ }</td>
</tr>
<tr>
<td>F → (E)</td>
<td>id</td>
<td>{ (, id }</td>
</tr>
</tbody>
</table>

Another Example of Calculating FIRST and FOLLOW

<table>
<thead>
<tr>
<th>Production</th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>X → Ya</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>Y → ZW</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>W → c</td>
<td>ε</td>
<td>{ }</td>
</tr>
<tr>
<td>Z → a</td>
<td>bZ</td>
<td>{ }</td>
</tr>
</tbody>
</table>

Constructing Predictive Parsing Tables

For each A → α do
1. Add A → α to M[A, a] for each a in FIRST(α)
2. If ε is in FIRST(α)
   a. Add A → α to M[A, b] for each b in FOLLOW(A)
   b. If $ is in FOLLOW(A) add A → α to M[A, $]
3. Make each undefined entry of M an error.

LL(1)

First "L" - scans input from left to right
Second "L" - produces a leftmost derivation
1 - uses one input symbol of lookahead at each step to make a parsing decision

A grammar whose predictive parsing table has no multiply-defined entries is LL(1).

No ambiguous or left-recursive grammar can be LL(1).
When Is a Grammar LL(1)?

A grammar is LL(1) iff for each set of productions where \( A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \), the following conditions hold.

1. \( \text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset \)
   where \( 1 \leq i \leq n \) and \( 1 \leq j \leq n \) and \( i \neq j \)

2. If \( \alpha_i \Rightarrow \varepsilon \) then
   a. \( \alpha_1, \ldots, \alpha_{i-1}, \alpha_i+1, \ldots, \alpha_n \) does not \( \Rightarrow \varepsilon \)
   b. \( \text{FIRST}(\alpha_i) \cap \text{FOLLOW}(A) = \emptyset \)
   where \( j \neq i \) and \( 1 \leq j \leq n \)

Checking If a Grammar is LL(1)

<table>
<thead>
<tr>
<th>Production</th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \to iEteSS' \mid a )</td>
<td>{ i, a }</td>
<td>{ e, $ }</td>
</tr>
<tr>
<td>( S' \to eS \mid \varepsilon )</td>
<td>{ e, \varepsilon }</td>
<td>{ e, $ }</td>
</tr>
<tr>
<td>( E \to b )</td>
<td>{ b }</td>
<td>{ t }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>a</th>
<th>b</th>
<th>e</th>
<th>i</th>
<th>t</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td></td>
<td></td>
<td></td>
<td>S\to a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S' )</td>
<td></td>
<td>S\to eS</td>
<td></td>
<td>S'\to \varepsilon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( E )</td>
<td></td>
<td>E\to b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So this grammar is not LL(1).

Bottom-Up Parsing

- Bottom-up parsing
  - attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root
  - is the process of reducing the string \( w \) to the start symbol of the grammar
  - at each step, we need to decide
    - when to reduce
    - what production to apply
    - actually, constructs a right-most derivation in reverse

Shift-Reduce Parsing

- Shift-reduce parsing is bottom-up.
  - A handle is a substring that matches the rhs of a production.
  - A shift moves the next input symbol on a stack.
  - A reduce replaces the rhs of a production that is found on the stack with the nonterminal on the left of that production.
  - A viable prefix is the set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser
Model of an LR Parser

- Each $S_i$ is a state.
- Each $X_j$ is a grammar symbol (when implemented these items do not appear in the stack).
- Each $a_i$ is an input symbol.
- All LR parsers can use the same algorithm (code).
- The action and goto tables are different for each LR parser.

LR (k) Parsing (cont.)

If config == $(s_0, X_1, s_1, X_2, s_2, ..., X_m, s_m, a_i, a_{i+1}, ..., a_n, \$)$
1. if action $[s_m, a_i] == \text{shift } s$ then
   new config is $(s_0, X_1, s_1, X_2, s_2, ..., X_m, s_m, a_i, a_{i+1}, ..., a_n, \$)$
2. if action $[s_m, a_i] == \text{reduce } A \rightarrow \beta$ and
   goto $[s_{m-r}, A] == s$ (where $r$ is the length of $\beta$) then
   new config is $(s_0, X_1, s_1, X_2, s_2, ..., X_{m-r}, s_{m-r}, A_s, a_i, a_{i+1}, ..., a_n, \$)$
3. if action $[s_m, a_i] == \text{ACCEPT}$ then stop
4. if action $[s_m, a_i] == \text{ERROR}$ then attempt recovery
Can resolve some shift-reduce conflicts with lookahead.
ex: LR(1)
Can resolve others in favor of a shift.
ex: $S \rightarrow iCtS | iCtSeS$

Advantages of LR Parsing

- LR parsers can recognize almost all programming language constructs expressed in context-free grammars.
- Efficient and requires no backtracking.
- Is a superset of the grammars that can be handled with predictive parsers.
- Can detect a syntactic error as soon as possible on a left-to-right scan of the input.

LR(k) Parsing

"L" - scans input from left to right
"R" - constructs a rightmost derivation in reverse
"k" - uses k symbols of lookahead at each step to make a parsing decision

Uses a stack of alternating states and grammar symbols. The grammar symbols are optional. Uses a string of input symbols ($\$ \text{ on end}$). Parsing table has an action part and a goto part.
LR Parsing Example

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → ( E )
6. F → id

LR Parsing Example

- It produces rightmost derivation in reverse:
  E → E + T → E + F → E + id
  → T + id → T * F + id
  → T * id + id → F * id + id
  → id * id + id

Calculating the Sets of LR(0) Items

LR(0) item - production with a dot at some position in the right side
Example:
A → BC has 3 possible LR(0) items
  A → B·C
  A → B·C
  A → B·C·
A → ε has 1 possible item
  A → ·

3 operations required to construct the sets of LR(0) items:
(1) closure, (2) goto, and (3) augment
Calculating Goto of a Set of LR(0) Items

Calculate goto \((I, X)\) where \(I\) is a set of items and \(X\) is a grammar symbol.

Take the closure (the set of items of the form \(A \rightarrow \alpha X \cdot \beta\)) where \(A \rightarrow \alpha \cdot X \cdot \beta\) is in \(I\).

**Grammar**

<table>
<thead>
<tr>
<th>Rule (\rightarrow)</th>
<th>(Goto) ((I_1,+)) for (I_1 = {E' \rightarrow E \cdot , E \rightarrow E \cdot + T})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E' \rightarrow E)</td>
<td>(E \rightarrow E \cdot + T)</td>
</tr>
<tr>
<td>(E \rightarrow E + T \mid T)</td>
<td>(T \rightarrow T \cdot F)</td>
</tr>
<tr>
<td>(T \rightarrow T \cdot F \mid F)</td>
<td>(T \rightarrow F)</td>
</tr>
<tr>
<td>(F \rightarrow ( E ) \mid id)</td>
<td>(F \rightarrow \cdot( E ))</td>
</tr>
<tr>
<td></td>
<td>(F \rightarrow \cdot id)</td>
</tr>
</tbody>
</table>

**Grammar**

<table>
<thead>
<tr>
<th>Rule (\rightarrow)</th>
<th>(Goto) ((I_2,\ast)) for (I_2 = {E \rightarrow T \cdot , T \rightarrow T \cdot * F})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T \rightarrow T \cdot * F)</td>
<td>(T \rightarrow F)</td>
</tr>
<tr>
<td>(F \rightarrow \cdot( E ))</td>
<td>(F \rightarrow \cdot id)</td>
</tr>
</tbody>
</table>

Augmenting the Grammar

- Given grammar \(G\) with start symbol \(S\), then an augmented grammar \(G'\) is \(G\) with a new start symbol \(S'\) and new production \(S' \rightarrow S\).

Analogy of Calculating the Set of LR(0) Items with Converting an NFA to a DFA

- Constructing the set of items is similar to converting an NFA to a DFA
  - each state in the NFA is an individual item
  - the closure \((I)\) for a set of items is the same as the \(\varepsilon\)-closure of a set of NFA states
  - each set of items is now a DFA state and goto \((I, X)\) gives the transition from \(I\) on symbol \(X\)

Sets of LR(0) Items Example

<table>
<thead>
<tr>
<th>Rule (\rightarrow)</th>
<th>(S \rightarrow L = R \mid R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(L \rightarrow *R \mid id)</td>
<td>(R \rightarrow L)</td>
</tr>
</tbody>
</table>
Constructing SLR Parsing Tables

Let \( C = \{I_0, I_1, \ldots, I_n\} \) be the parser states.

1. If \([A \rightarrow \alpha \cdot a \beta] \) is in \( I_i \) and goto \((I_i, a) = I_j\) then set action \([i, a]\) to 'shift j'.

2. If \([A \rightarrow \alpha \cdot] \) is in \( I_i \), then set action \([i, \alpha]\) to 'reduce \( A \rightarrow \alpha \)' for all \( \alpha \) in the FOLLOW(\( A \)). A may not be \( S' \).

3. If \([S' \rightarrow S \cdot] \) is in \( I_i \), then set action \([i, \$]\) to 'accept'.

4. If goto \((I_i, A) = I_j\), then set goto \([i, A]\) to \( j \).

5. Set all other table entries to 'error'.

6. The initial state is the one holding \([S' \rightarrow \cdot S]\).

LR (1) (cont.)

Solution - split states by adding LR(1) lookahead

\( [A \rightarrow \alpha \cdot \beta, a] \)

where \( A \rightarrow \alpha \beta \) is a production and 'a' is a terminal or endmarker $.

Closure(I) is now slightly different

repeat

for each item \([A \rightarrow \alpha \cdot B \beta, a] \) in I,

each production \( B \rightarrow \gamma \) in the grammar,

and each terminal \( b \) in FIRST(\( \beta \alpha \)) do

add \([B \rightarrow \gamma, b]\) to I (if not there)

until no more items can be added to I

Start the construction of the set of LR(1) items by computing the closure of \( \{[S' \rightarrow \cdot S, \$]\} \).

LR(1)

The unambiguous grammar

\[
\begin{align*}
S & \rightarrow L = R | R \\
L & \rightarrow *R | id \\
R & \rightarrow L
\end{align*}
\]

is not SLR.

See Fig 4.39.

action\([2, =]\) can be a "shift 6" or "reduce \( R \rightarrow L\)"

FOLLOW(\( R \)) contains "=" but no form begins with "\( R=\)".

**LR(1) Example**

\[
\begin{align*}
(0) & 1. S' \rightarrow S \\
(1) & 2. S \rightarrow CC \\
(2) & 3. C \rightarrow cC \\
(3) & 4. C \rightarrow d \\
\end{align*}
\]

\[
\begin{align*}
I_0: & \quad [S' \rightarrow S, \$] \quad \text{goto (\$)} = I_1 \\
& \quad [S \rightarrow CC, \$] \quad \text{goto (C)} = I_2 \\
& \quad [C \rightarrow cC, c/d] \quad \text{goto (c)} = I_3 \\
& \quad [C \rightarrow d, c/d] \quad \text{goto (d)} = I_4 \\
I_1: & \quad [S' \rightarrow S, \$] \\
I_2: & \quad [S \rightarrow C \cdot C, \$] \quad \text{goto (C)} = I_5 \\
& \quad [C \rightarrow cC, \$] \quad \text{goto (c)} = I_6 \\
& \quad [C \rightarrow d, \$] \quad \text{goto (d)} = I_7
\end{align*}
\]
LR(1) Example (cont.)

I₃: [C → c·C, c/d] goto ( C ) = I₈
[ C → ·cC, c/d ] goto ( c ) = I₃
[ C → ·d, c/d ] goto ( d ) = I₄

I₄: [ C → d·, c/d ]
I₅: [ S → CC·, $ ]
I₆: [ C → c·C, $ ] goto ( C ) = I₉
[ C → ·cC, $ ] goto ( c ) = I₆
[ C → ·d, $ ] goto ( d ) = I₇

I₇: [ C → d·, $ ]
I₈: [ C → cC·, c/d ]
I₉: [ C → cC·, $ ]

Constructing the LR(1) Parsing Table

Let C = {I₀, I₁, ..., Iₙ}

1. If [ A → α·aβ ] is in Iᵢ and goto(Iᵢ, a) = Iⱼ then set action[i, a] to “shift j”.
2. If [ A → α·, a ] is in Iᵢ then set action[i, a] to ‘reduce A→α’. A may not be S’.
3. If [ S’→S, $ ] is in Iᵢ then set action[i, $ ] to “accept.”
4. If goto(Iᵢ, A) = Iⱼ then set goto[i, A] to j.
5. Set all other table entries to error.
6. The initial state is the one holding [S’→·S, $]

Constructing LALR Parsing Tables

• Combine LR(1) sets with the same sets of the first parts (ignore lookahead).
• Table is the same size as SLR.
• Will not introduce shift-reduce conflicts because shifts don't use lookahead.
• May introduce reduce-reduce conflicts but seldom do for programming languages.

Last example collapses to table shown in Fig 4.41.

Using Ambiguous Grammars

1. E → E + E
2. E → E * E instead of T → T * F | F
3. E → ( E ) F → ( E ) | id
4. E → id

See Figure 4.48.

Advantages:
Grammar is easier to read.
Parser is more efficient.
Using Ambiguous Grammars (cont.)

Can use precedence and associativity to solve the problem.

See Fig 4.49.

shift / reduce conflict in state action[7,+]=(s4,r1)

s4 = shift 4 or E → E + E
r1 = reduce 1 or E → E + E.

id + id + id
↑ cursor here

action[7,*]=(s5,r1)
action[8,+]=(s4,r2) action[8,*]=(s5,r2)

Another Ambiguous Grammar

0. S’ → S
1. S → iSeS
2. S → iS
3. S → a

See Figure 4.50.

action[4,e]=(s5,r2)

followed by Fig. 4.49
followed by Fig. 4.50, 4.51

Ambiguities from Special-Case Productions

E → E sub E sup E
E → E sub E
E → E sup E
E → { E }
E → c

Ambiguities from Special-Case Productions (cont)

1. E → E sub E sup E
2. E → E sub E
3. E → E sup E
4. E → { E }
5. E → c

FIRST(E) = { ',', c }
FOLLOW(E) = {sub, sup, ',', $}

sub, sup have equal precedence and are right associative
Ambiguities from Special-Case Productions (cont)

1. $E \rightarrow E \, \text{sub} \, E \, \text{sup} \, E$
   \[ \text{FIRST}(E) = \{ '{', 'c' \} \]

2. $E \rightarrow E \, \text{sub} \, E$
   \[ \text{FOLLOW}(E) = \{ \text{sub, sup, '}$, '$' \} \]

3. $E \rightarrow E \, \text{sup} \, E$

4. $E \rightarrow \{ E \}$
   \[ \text{sub, sup have equal precedence and are right associative} \]

5. $E \rightarrow c$

YACC Declarations

- In declarations:
  - Can put ordinary C declarations in
    \%
    \{
    ...
    \%
  - Can declare tokens using
    \%
    \%token
    \%left
    \%right
  - Precedence is established by the order the operators are listed (low to high).

YACC Translation Rules

- Form
  \[ A : \text{Body} ; \]
  where $A$ is a nonterminal and $\text{Body}$ is a list of nonterminals and terminals.

- Semantic actions can be enclosed before or after each grammar symbol in the body.

- Yacc chooses to shift in a shift/reduce conflict.

- Yacc chooses the first production in a reduce/reduce conflict.
Yacc Translation Rules (cont.)

- When there is more than one rule with the same left hand side, a '|' can be used.

```
A : B C D ;
A : E F ;
A : G ;
=>
A : B C D | E F | G ;
```

Yacc Actions

- Actions are C code segments enclosed in {} and may be placed before or after any grammar symbol in the right hand side of a rule.
- To return a value associated with a rule, the action can set $$.
- To access a value associated with a grammar symbol on the right hand side, use $i$, where i is the position of that grammar symbol.
- The default action for a rule is
  ```
  { $$ = $1; }
  ```

Example of a Yacc Specification

```
%token IF ELSE NAME /* defines multicharacter tokens */
%right '=' /* low precedence, a=b=c shifts */
%left '+' '-' /* mid precedence, a-b-c reduces */
%left '*' '/' /* high precedence, a/b/c reduces */

%%

stmt : expr ';' |
  IF '(' expr ')' stmt |
  IF '(' expr ')' stmt ELSE stmt; /* prefers shift to reduce in shift/reduce conflict */

expr : NAME '=' expr /* assignment */ |
  expr '+' expr |
  expr '-' expr |
  expr '*' expr |
  expr '/' expr |
  '-' expr  %prec '*' /* can override precedence */ |
  NAME ;

%% /* definitions of yylex, etc. can follow */
```

Syntax Error Handling

- Errors can occur at many levels
  - lexical - unknown operator
  - syntactic - unbalanced parentheses
  - semantic - variable never declared
  - logical - dereference a null pointer
- Goals of error handling in a parser
  - detect and report the presence of errors
  - recover from each error to be able to detect subsequent errors
  - should not slow down the processing of correct programs
Syntax Error Handling (cont.)

- Viable-prefix property - detect an error as soon as see a prefix of the input that is not a prefix of any string in the language.

Error-Recovery Strategies

- Panic- mode
  - skip until one of a synchronizing set of tokens is found (e.g. ';', 'end'). Is very simple to implement but may miss detection of some error (when more than one error in a single statement)

- Phase- level
  - replace prefix of remaining input by a string that allows the parser to continue. Hard for the compiler writer to anticipate all error situations

Error-Recovery Strategies (cont...)

- Error productions
  - augment the grammar of the source language to include productions for common errors. When production is used, an appropriate error diagnostic would be issued. Feasible to only handle a limited number of errors.

- Global correction
  - choose minimal sequence of changes to allow a least-cost correction. Too costly to actually be implemented in a parser. Also the closest correct program may not be what the programmer intended.

Error-Recovery in Predictive Parsing

- It is easier to recover from an error in a nonrecursive predictive parser than using recursive descent.

- Panic- mode recovery
  - assume the nonterminal A is on the stack when we encounter an error. As a starting point can place all symbols in FOLLOW(A) into the synchronizing set for the nonterminal A. May also wish to add symbols that begin higher constructs to the synchronizing set of lower constructs. If a terminal is on top of the stack, then can pop the terminal and issue a message stating that the terminal was discarded.
Error-Recovery in Predictive Parsing (cont.)

- Phrase-level recovery
  - can be implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. The compiler writer would attempt each situation appropriately (issue error message and update input symbols and pop from the stack).

Error-Recovery in LR Parsing (cont)

- Phrase-level recovery
  - implement an error recovery routine for each error entry in the table.

- Error productions (Used in YACC)
  - pops symbols until topmost state has an error production, then shifts error onto stack. Then discards input symbols until it finds one that allows parsing to continue. The semantic routine with an error production can just produce a diagnostic message.

Error-Recovery in LR Parsing

- Canonical LR Parser
  - will never make a single reduction before recognizing an error.

- SLR & LALR Parsers
  - may make extra reductions but will never shift an erroneous input symbol on the stack.

- Panic-mode recovery
  - scan down stack until a state with a goto on a particular nonterminal representing a major program construct (e.g. expression, statement, block, etc.) is found. Input symbols are discarded until one is found that is in the FOLLOW of the nonterminal. The parser then pushes on the state in goto. Thus, it attempts to isolate the phase containing the error.