The two grammars generate the same language, but the one on the right generates the \( \beta \) first, and then a string of \( \alpha \)'s, using a rule that is right recursive instead of left recursive.
First sets

\[ \text{First}(\alpha) = \begin{cases} \{ \alpha \} & \text{if } \alpha \in \Sigma \\ \cup_{(\alpha \rightarrow \omega_i) \in P} \text{First}(\omega_i) & \text{if } \alpha \in V \\ \{ \lambda \} & \text{if } \alpha = \lambda \end{cases} \]

\[ \text{First}(\alpha_1 \ldots \alpha_L) = \bigcup_{j=1}^{L} \text{First}(\alpha_j) \]

<table>
<thead>
<tr>
<th>A \rightarrow BC</th>
<th>\omega \rightarrow \text{First}(\omega)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H \rightarrow { h, \lambda }</td>
</tr>
<tr>
<td></td>
<td>G \rightarrow { g }</td>
</tr>
<tr>
<td></td>
<td>C \rightarrow { c, \lambda }</td>
</tr>
<tr>
<td></td>
<td>B \rightarrow { b }</td>
</tr>
<tr>
<td></td>
<td>E \rightarrow { e, \lambda }</td>
</tr>
<tr>
<td></td>
<td>F \rightarrow { c, e, \lambda }</td>
</tr>
<tr>
<td></td>
<td>A \rightarrow { b, e, c, g, h, \lambda }</td>
</tr>
<tr>
<td></td>
<td>BC \rightarrow { b }</td>
</tr>
<tr>
<td></td>
<td>EFGH \rightarrow { e, c, g }</td>
</tr>
</tbody>
</table>

Follow sets

1. Initially set \( \text{Follow}(N) = \emptyset, \forall N \in V \).
2. Given production \( A \rightarrow \alpha B \beta \), set

\[ \text{Follow}(B) = \text{Follow}(B) \cup (\text{First}(\beta) \setminus \{ \lambda \}) \]

3. Given production \( A \rightarrow \alpha B \beta \), where \( \lambda \in \text{First}(\beta) \), set

\[ \text{Follow}(B) = \text{Follow}(B) \cup \text{Follow}(A) \]

<table>
<thead>
<tr>
<th>A \rightarrow BC</th>
<th>\text{Follow}(N)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A \rightarrow { }</td>
</tr>
<tr>
<td></td>
<td>B \rightarrow \text{First}(C) \cup \text{Follow}(A) = { c }</td>
</tr>
<tr>
<td></td>
<td>F \rightarrow \text{First}(G) = { g }</td>
</tr>
<tr>
<td></td>
<td>C \rightarrow \text{Follow}(A) \cup \text{First}(E) \cup \text{Follow}(F) = { e, g }</td>
</tr>
<tr>
<td></td>
<td>E \rightarrow \text{First}(F) \cup \text{First}(G) = { c, e, g }</td>
</tr>
<tr>
<td></td>
<td>G \rightarrow \text{Follow}(H) \cup \text{Follow}(A) = { h }</td>
</tr>
<tr>
<td></td>
<td>H \rightarrow \text{Follow}(A) = { }</td>
</tr>
</tbody>
</table>
Recursive descent parser generation

Procedure NonTermN
    if (LookAhead() ∈ First(ω₁), where (N → ω₁) ∈ P) then
        /* Use ω₁ to generate calls to Expect() and other nonterminals*/
    else
        if (LookAhead() ∈ Follow(N) and (N → λ) ∈ P) then
            return ()
        else
            /* error*/
    fi
    fi
end

Recursive descent – Example

\[
\begin{align*}
S & \rightarrow A \ C \ \$ \\
C & \rightarrow c \\
& \mid \ \lambda \\
A & \rightarrow a \ B \ C \ d \\
& \mid BQ \\
& \mid \ \lambda \\
B & \rightarrow b \ B \\
& \mid d \\
Q & \rightarrow q
\end{align*}
\]

First | Follow
--- | ---
S | \{ a, b, d, c, $\} | \{ \}
A | \{ a, b, d, \lambda \} | \{ c, $\}
B | \{ b, d \} | \{ c, d, q \}
C | \{ c, \lambda \} | \{ d, $\}
Q | \{ q \} | \{ c, $\}
The generated procedures

\[
\begin{align*}
S & \rightarrow A C \, \$ \\
C & \rightarrow c \\
A & \rightarrow a \, B C \, d \\
B & \rightarrow b \, B \\
Q & \rightarrow q
\end{align*}
\]

Procedure \( S() \)

\[
\text{if}(\text{Look Ahead}(\cdot) \in \{ a, b, d, c, \$ \}) \text{ then} \\
\quad \text{call A()} \\
\quad \text{call C()} \\
\quad \text{call Expect(\$)} \\
\text{else} \\
\quad /* \text{error}*/
\]

fi
end

Procedure \( C() \)

\[
\text{if}(\text{Look Ahead}(\cdot) \in \{ c \}) \text{ then} \\
\quad \text{call Expect(c)} \\
\text{else} \\
\quad \text{if}(\text{Look ahead}(\cdot) \notin \{ d, \$ \}) \text{ then} \\
\quad \quad /* \text{error}*/
\]

fi
fi
end

The generated procedures (cont’d)

\[
\begin{align*}
S & \rightarrow A C \, \$ \\
C & \rightarrow c \\
A & \rightarrow a \, B C \, d \\
B & \rightarrow b \, B \\
Q & \rightarrow q
\end{align*}
\]

Procedure \( A() \)

\[
\text{if}(\text{Look Ahead}(\cdot) \in \{ a \}) \text{ then} \\
\quad \text{call Expect(a)} \\
\quad \text{call B()} \\
\quad \text{call C()} \\
\quad \text{call Expect(d)} \\
\text{else} \\
\quad \text{if}(\text{Look Ahead}(\cdot) \in \{ b, d \}) \text{ then} \\
\quad \quad \text{call B()} \\
\quad \quad \text{call Q()} \\
\quad \text{else} \\
\quad \quad \text{if}(\text{Look Ahead}(\cdot) \in \{ c, \$ \}) \text{ then} \\
\quad \quad \quad \text{return ()} \\
\quad \quad \text{else} \\
\quad \quad \quad /* \text{error}*/
\]

fi
fi
end

Procedure \( B() \)

\[
\text{call Q()} \\
\text{else} \\
\quad \text{if}(\text{Look Ahead}(\cdot) \in \{ c, \$ \}) \text{ then} \\
\quad \quad \text{return ()} \\
\quad \text{else} \\
\quad \quad /* \text{error}*/
\]

fi
fi
end

Procedure \( Q() \)

\[
\text{call Q()} \\
\text{else} \\
\quad /* \text{error}*/
\]

fi
fi
end
The generated procedures (cont'd)

<table>
<thead>
<tr>
<th></th>
<th>First</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>a, b, d, c, $</td>
<td>{ }</td>
</tr>
<tr>
<td>C</td>
<td>a, b, d, λ</td>
<td>{ c, $ }</td>
</tr>
<tr>
<td>A</td>
<td>b, d</td>
<td>{ c, d, q }</td>
</tr>
<tr>
<td>B</td>
<td>c, λ</td>
<td>{ d, $ }</td>
</tr>
<tr>
<td>Q</td>
<td>q</td>
<td>{ c, $ }</td>
</tr>
</tbody>
</table>

Procedure B()

```plaintext
if (LookAhead(\[ b \]) \) then
    call Expect(b)
    call B()
else
    if (LookAhead(\[ d \]) \) then
        call Expect(d)
    else
        /* error */
        fi
fi
fi
end
```

Procedure Q()

```plaintext
if (LookAhead(\[ q \]) \) then
    call Expect(q)
else
    /* error */
    fi
end
```

Recursive descent – expression grammar

<table>
<thead>
<tr>
<th></th>
<th>First</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>a, a</td>
<td>{ }, $</td>
</tr>
<tr>
<td>E'</td>
<td>+, -</td>
<td>{ }, $</td>
</tr>
<tr>
<td>T</td>
<td>a, a</td>
<td>{ +, -, $ }</td>
</tr>
<tr>
<td>T'</td>
<td>*, /</td>
<td>{ +, -, $ }</td>
</tr>
<tr>
<td>F</td>
<td>a, a</td>
<td>{ *, /, +, -, $ }</td>
</tr>
</tbody>
</table>

Procedure E'

```plaintext
if (LookAhead(+) \) then
    call Expect( +)
    call T
    call E'
else
    if (LookAhead(-) \) then
        call Expect(-)
        call T
        call E'
    else
        if (LookAhead($, ',) \) then
            return()
        else
            call Error()
        fi
    fi
fi
end
```
Maintaining lookahead

Procedure main()
    LAtok ← GetNextToken()
    call S()
end

Function LookAhead() : token
    return (LAtok)
end

Procedure Expect(tok)
    if (LAtok = tok) then
        LAtok ← GetNextToken()
    else
        /* error */
    fi
end

A lookahead of \( k \) tokens is maintained by appropriately buffering the input.

Technically, \( k \) lookahead is equivalent in power to a single token of lookahead. The proof is constructive: each permutation of \( k \) symbols is encoded as a single token.

The Expect(tok) procedure first compares the incoming token against tok, and then advances input into the lookahead buffer.

Copyright © 1994 Ron K. Cytron. All rights reserved

Recursive descent – correctness and properties

When is our recursive descent parser construction successful? If the grammar involves any left-recursion, then our construction method will create a parser containing an infinite loop. So, we require that the grammar be free of left-recursion.

The grammar transformation technique covered earlier can help eliminate left-recursion.

Also, we require that the parser operate deterministically: actions taken at each step make progress toward completion, so that backtracking is not necessary.

Thus, given a set of rules for nonterminal \( N \)

\[
N \rightarrow \omega_1 \\
\vdots \\
\omega_n
\]

we require

1. \[ \cap_i \text{First}(\omega_i) = \{ \} \]

2. If \( \lambda = \omega_j, 1 \leq j \leq n \), then we also require
\[
\cup_i (\text{Follow}(N) \cap \text{First}(\omega_i)) = \{ \}
\]
Error repair

Good programming languages are designed with a relatively large "distance" between syntactically correct programs, to increase the likelihood that conceptual mistakes are caught as syntactic errors.

Error repair usually occurs at two levels:

Local: repairs mistakes with little global import, such as missing semicolons and undeclared variables.

Scope: repairs the program text so that scopes are correct. Errors of this kind include unbalanced parentheses and begin/end blocks.

Repair actions can be divided into insertions and deletions. Typically the compiler will use some lookahead and backtracking in attempting to make progress in the parse. There is great variation among compilers, though some languages (PL/C) carry a tradition of good error repair. Goals of error repair include:

1. No input should cause the compiler to collapse.
2. Illegal constructs are flagged.
3. Frequently occurring errors are repaired gracefully.
4. Minimal stuttering or cascading of errors.

LL-style parsing lends itself well to error repair, since the compiler uses the grammar’s rules to predict what should occur next in the input.
Augmenting recursive descent parsers for error recovery

Recursive and LL parsers are often called *predictive*, because they operate by predicting the next step in a derivation.

Suppose the parser is operating in procedure *A* for some nonterminal *A*. If an error occurs, it seems reasonable to recover by skipping to a symbol that could follow *A*, and then return.

```
E  →  TE'
E' →  +TE'
E' →  −TE'
T  →  FT'
T' →  *FT'
T' →  /FT'
F  →  (E)
|   a

Procedure E(StopSet)

if (LookAhead(+)) then
  call Expect(+)
  call T(\{+,−\} ∪ StopSet)
  call E(StopSet)
else
  if (LookAhead(\$, ’)') then
    return ()
  else
    call ErrorRecover(StopSet)
```

```
First    Follow
E  \{(a\,\,)}    \{\,\}$
E' \{+,−\}    \{\,\}$
T  \{(a\,\,)}    \{+,−,\,\}$
T' \{*,/\}    \{+,−,\,\}$
F  \{(a\,\,)}    \{*,/+,−,\,\}$
```

Table-driven \(LL(k)\) parsing

Our recursive descent parser contained a procedure for each nonterminal. The generation of these procedures could be automated—through the construction and testing of *First* and *Follow* sets—for any grammar free of left recursion.

Another equally automatable approach is to use a simple parsing engine that is driven by tables constructed by similar analysis of the grammar.

The parsing engine begins by pushing the start symbol *S* onto the stack. Each subsequent action is one of the following:

**Match**: pairs an input symbol *a* an *a* on top-of-stack.

**Apply**: replaces the nonterminal *N* with \(\omega\), where \((N \rightarrow \omega) \in P\).
Match

If the top-of-stack contains the terminal symbol “a”, then the parsing engine must find an “a” as the next input symbol; the stack is popped, and the input is advanced.

Before  a

After

- If a match simultaneously empties the stack and exhausts the input stream, then the string is accepted by the parser.
- If a match is attempted, but the symbols disagree, then an error is declared.

Apply

If the top-of-stack contains a nonterminal $N$, then the parsing engine must choose the appropriate rule for $N$, say $N \rightarrow \alpha \beta \gamma$. The stack is popped of symbol $N$, and the symbols $\alpha$, $\beta$, and $\gamma$ are pushed onto the stack, such that $\alpha$ is the new top-of-stack.

Before

\[
\begin{array}{c}
N \\
\alpha \\
\beta \\
\gamma \\
\end{array}
\]

After

\[
\begin{array}{c}
a \\
\alpha \\
\beta \\
\gamma \\
\end{array}
\]

Since a match is always required when a terminal is exposed on top-of-stack, the only information that must be coded in our table is the rule that should be applied when a nonterminal appears on top-of-stack. As with our recursive descent parser, this decision can be based on $k$ symbols of lookahead into the input stream.
## Constructing the table

<table>
<thead>
<tr>
<th>NonTerm</th>
<th>Lookahead</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
<th>( q )</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>( C )</td>
<td></td>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( A )</td>
<td></td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>( B )</td>
<td></td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( Q )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>

The nonblank entries in the above table indicate the number of the rule that should be applied, given a nonterminal on top-of-stack and an input symbol as lookahead.

## Using the table

<table>
<thead>
<tr>
<th>NonTerm</th>
<th>Lookahead</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
<th>( q )</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>( C )</td>
<td></td>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( A )</td>
<td></td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>( B )</td>
<td></td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( Q )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>

Below is shown the stack activity in parsing the input string “abbddc$”.

```
Input string
  a b b d d c $
```

```plaintext
  a B B B d
  C C C C C C C C $                     Input string
  A d d d d d
  C C C C C C C C c
  S $ $ $ $ $ $ $
  1 4 7 7 8 3 2
```
Like the top-down parser, the bottom-up parser checks for errors on a shift. The parse table we shall construct indicates when a shift is error-free.

Actually, instead of pushing a symbol onto the stack, we push a state, which indexes the parse table and represents the current possibilities of the parse.
• If the rule applied is $N \rightarrow \omega$, where $\omega$ has $m$ symbols, then $m$ symbols are popped off the stack, and a symbol representing $N$ is pushed.
• It’s important to remember that a canonical parse can perform reductions only at the top-of-stack.

This is LR-style parsing: a scan from the left that produces a rightmost derivation.
We could have tried to apply $C \rightarrow \lambda$ at any point during the parse, but most would not have made progress toward an accept. Where parse table construction is successful, the table directs the parse towards an accept if one is possible.
LR table construction

Each state of the parser represents parsing possibilities after processing a given prefix of the input string.

To construct the canonical \( LR(0) \) set of states:

1. Each state begins with a *kernel* that represents progress through certain rules of the grammar:

\[
\begin{align*}
(3) & 
X & \rightarrow & \ y \cdot \ z \\
W & \rightarrow & \ x \, z \, y \cdot A \\
F & \rightarrow & \ a \, B \, C \, y \cdot \\
\end{align*}
\]

The dot (*) shows the progress through the rule achieved by moving into this state.

2. When \( * \) is next to a nonterminal, we must add into this state the closure by expanding all rules of the nonterminal:

\[
\begin{align*}
(3) & 
A & \rightarrow & \ b \, c \, d \\
A & \rightarrow & \ z \, A \\
\end{align*}
\]

We then label each component of the state with an action, indicating transfer to some other state, reduction by a rule, or accept:

\[
\begin{align*}
(3) & 
X & \rightarrow & \ y \cdot \ z & \text{Goto State} & 17 \\
W & \rightarrow & \ x \, z \, y \cdot A & \text{Goto State} & 5 \\
F & \rightarrow & \ a \, B \, C \, y \cdot & \text{Reduce by rule} & 5 \\
A & \rightarrow & \ b \, c \, d & \text{Goto State} & 2 \\
A & \rightarrow & \ z \, A & \text{Goto State} & 17 \\
\end{align*}
\]

which may create a new state:

\[
\begin{align*}
(17) & 
X & \rightarrow & \ y \cdot z \cdot & \text{Reduce by rule} & 10 \\
A & \rightarrow & \ z \cdot A & \text{Goto State} & 1 \\
A & \rightarrow & \ b \, c \, d & \text{Goto State} & 2 \\
A & \rightarrow & \ z \, A & \text{Goto State} & 18 \\
\end{align*}
\]
Conflict resolution

Within a state, how do we resolve whether to shift or reduce when either action seems appropriate?

Examining the Follow information shows that only those input symbols in \{c, $\} can follow an A. In state (1) we therefore reduce only when “c” or “$” appears next in the input. Since these symbols are disjoint from the input symbols that cause shifts into other states (\{a, b, d\}), we can resolve the apparent conflict.

In general, a state might have an apparent shift/reduce or reduce/reduce conflict. The more expensive table construction methods generally provide better conflict resolution.

Table for our example

<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>q</th>
<th>$</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>
Using the table

<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>q</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td>12</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td></td>
<td>14</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Using the table (cont’d)

<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>q</th>
<th>S</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td>12</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td></td>
<td>14</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Set of items construction for our expression grammar

<table>
<thead>
<tr>
<th>Rule</th>
<th>First</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. S → E$</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>2. E → E + T</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>3. T → T* F</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>4. F → (E)</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>5. a</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

The above shift/reduce conflict is resolved by noting that * ∉ Follow(E).

Copyright © 1994 Ron K. Cytron. All rights reserved ±6 7± SIGPLAN '94 COMPILER CONSTRUCTION TUTORIAL
### The resulting parse table

<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
<th>E</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>Goto State 1</td>
<td>5</td>
<td>Goto State 5</td>
<td>2</td>
<td>Goto State 2</td>
<td>3</td>
<td>Goto State 3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Goto State 8</td>
<td>9</td>
<td>Goto State 9</td>
<td>7</td>
<td>Goto State 7</td>
<td>3</td>
<td>Goto State 3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Reduce by rule 5</td>
<td>Reduce by rule 3</td>
<td>Reduce by rule 3</td>
<td>Reduce by rule 3</td>
<td>Reduce by rule 5</td>
<td>Reduce by rule 5</td>
<td>Reduce by rule 5</td>
<td>Reduce by rule 5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>Goto State 6</td>
<td>5</td>
<td>Goto State 5</td>
<td>10</td>
<td>Goto State 10</td>
<td>3</td>
<td>Goto State 3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Goto State 7</td>
<td>7</td>
<td>Goto State 7</td>
<td>4</td>
<td>Goto State 4</td>
<td>2</td>
<td>Goto State 2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Reduce by rule 4</td>
<td>Reduce by rule 4</td>
<td>Reduce by rule 4</td>
<td>Reduce by rule 4</td>
<td>Reduce by rule 4</td>
<td>Reduce by rule 4</td>
<td>Reduce by rule 4</td>
<td>Reduce by rule 4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>Goto State 6</td>
<td>5</td>
<td>Goto State 5</td>
<td>11</td>
<td>Goto State 11</td>
<td>4</td>
<td>Goto State 4</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>Reduce by rule 2</td>
<td>Goto State 2</td>
<td>Reduce by rule 2</td>
<td>Reduce by rule 2</td>
<td>Reduce by rule 2</td>
<td>Reduce by rule 2</td>
<td>Reduce by rule 2</td>
<td>Reduce by rule 2</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Reduce by rule 4</td>
<td>Reduce by rule 4</td>
<td>Reduce by rule 4</td>
<td>Reduce by rule 4</td>
<td>Reduce by rule 4</td>
<td>Reduce by rule 4</td>
<td>Reduce by rule 4</td>
<td>Reduce by rule 4</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Reduce by rule 6</td>
<td>Reduce by rule 6</td>
<td>Reduce by rule 6</td>
<td>Reduce by rule 6</td>
<td>Reduce by rule 6</td>
<td>Reduce by rule 6</td>
<td>Reduce by rule 6</td>
<td>Reduce by rule 6</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Reduce by rule 6</td>
<td>Reduce by rule 6</td>
<td>Reduce by rule 6</td>
<td>Reduce by rule 6</td>
<td>Reduce by rule 6</td>
<td>Reduce by rule 6</td>
<td>Reduce by rule 6</td>
<td>Reduce by rule 6</td>
<td></td>
</tr>
</tbody>
</table>

### Using the table

Using the table, we can construct the parse tree for the expression `a + a * (a + a)`. The parse tree is shown in the diagram below.
Using the table

Using the table