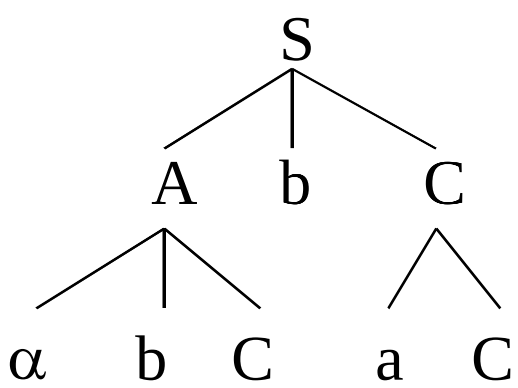
# UNIT-3

# Topics to be covered:

- Basic Parsing Techniques: Parsers, Shift reduce parsing, operator  precedence parsing, top down parsing, predictive parsers ,Automatic Construction of efficient Parsers: LR parsers, the canonical Collection of LR(0) items, constructing SLR,CLR & LALR parsing tables, using ambiguous grammars, an automatic parser generator, implementation of LR parsing tables.

# Shift-Reduce Parsers

- Reviewing some technologies:
  - Phrase
  - Simple phrase
  - Handle of a sentential form




A sentential form

$$\underbrace{\alpha\ b\ C}_{\textbf{handle}}\ b\ \underbrace{a\ C}_{\textbf{Simple phrase}}$$

# Shift-reduce parser

- A parse stack
  - Initially empty, contains symbols already parsed
    - Elements in the stack are not terminal or nonterminal symbols
  - The parse stack catenated with the remaining input always represents a right sentential form
  - Tokens are shifted onto the stack until the top of the stack contains the handle of the sentential form

# Shift-reduce parser

- Two questions
  1. Have we reached the end of handles and how long is the handle?
  2. Which nonterminal does the handle reduce to?
- We use tables to answer the questions
  - ACTION table
  - GOTO table

# Shift-reduce parser

- LR parsers are driven by two tables:
  - *Action table*, which specifies the actions to take
    - Shift, reduce, accept or error
  - *Goto table*, which specifies state transition
- We push states, rather than symbols onto the stack
- Each state represents the possible subtree of the parse tree

# Shift-reduce parser

1. <program> → **begin** <stmts> **end** $
2. <stmts> → SimpleStmt ; <stmts>
3. <stmts> → **begin** <stmts> **end** ; <stmts>
4. <stmts> → λ

| Symbol | State | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| **begin** | S | S | | | S | | S | | | S | | |
| **end** | | R4 | S | | R4 | | R4 | S | | R4 | R2 | R3 |
| ; | | | | | | S | | | S | | | |
| SimpleStmt | | S | | | S | | S | | | S | | |
| $ | | | | A | | | | | | | | |
| <program> | | | | | | | | | | | | |
| <stmts> | | S | | | S | | S | | | S | | |

**Figure 6.2** A Shift-Reduce `action` Table for $G_0$

| Symbol | State | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| **begin** | 1 | 4 | | | 4 | | 4 | | | 4 | | |
| **end** | | | 3 | | | | | 8 | | | | |
| ; | | | | | | 6 | | | 9 | | | |
| SimpleStmt | | 5 | | | 5 | | 5 | | | 6 | | |
| $ | | | | | | | | | | | | |
| <program> | | | | | | | | | | | | |
| <stmts> | | 2 | | | 7 | | 10 | | | 11 | | |

**Figure 6.3** A Shift-Reduce `go_to` Table for $G_0$

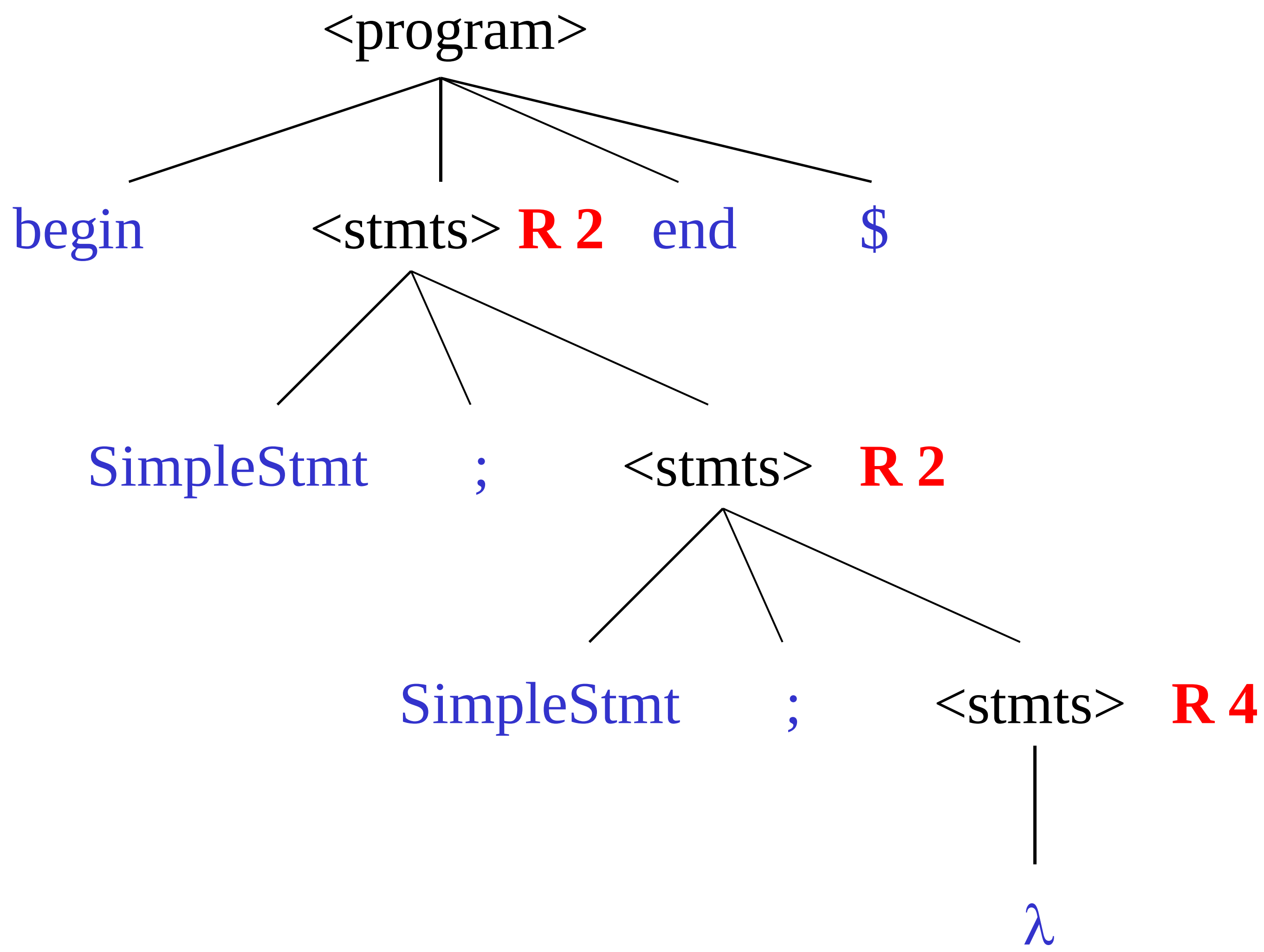
```
void shift_reduce_driver(void)
{
   /*
    * Push the Start State, S0,
    * onto an empty parse stack.
    */
   push(S0);

   while (TRUE) {   /* forever */
      /*
       * Let S be the top parse stack state;
       * let T be the current input token.
       */

      switch (action[S][T]) {
      case ERROR:
         announce_syntax_error();
         break;

      case ACCEPT:
         /* The input has been correctly parsed. */
         clean_up_and_finish();
         return;

      case SHIFT:
         push(go_to[S][T]);
         scanner(& T);   /* Get next token. */
         break;

      case Reduce_i:
         /*
          * Assume i-th production is X → Y1 · · · Ym.
          * Remove states corresponding to
          * the RHS of the production.
          */
         pop(m);
         /* S′ is the new stack top. */
         push(go_to[S′][X]);
         break;
      }
   }
}
```



**Figure 6.1** A Simple Shift-Reduce Driver

| Step | Parse Stack | Remaining Input | Parser Action |
|---|---|---|---|
| (1) | 0 | **begin** SimpleStmt ; SimpleStmt ; **end** $ | Shift |
| (2) | 0,1 | SimpleStmt ; SimpleStmt ; **end** $ | Shift |
| (3) | 0,1,5 | ; SimpleStmt ; **end** $ | Shift |
| (4) | 0,1,5,6 | SimpleStmt ; **end** $ | Shift |
| (5) | 0,1,5,6,5 | ; **end** $ | Shift |
| (6) | 0,1,5,6,5,6 | **end** $ | Reduce 4 |
| (7) | 0,1,5,6,5,6,10 | **end** $ | Reduce 2 |
| (8) | 0,1,5,6,10 | **end** $ | Reduce 2 |
| (9) | 0,1,2 | **end** $ | Shift |
| (10) | 0,1,2,3 | $ | Accept |

**Figure 6.4** Example of a Shift-Reduce Parse


<program>
begin
<stmts> R 2
end
$
SimpleStmt
;
<stmts> R 2
SimpleStmt
;
<stmts> R 4
λ

# LR Parsers

- LR(*1*):
  - left-to-right scanning
  - rightmost derivation(reverse)
  - *1*-token lookahead
- LR parsers are deterministic
  - no backup or retry parsing actions
- LR(*k*) parsers
  - decide the next action by examining the tokens already shifted and at most *k* lookahead tokens
  - the most powerful of deterministic bottom-up parsers with at most *k* lookahead tokens.

# LR(0) Parsing

- A production has the form
  - $A \to X_1 X_2 \ldots X_j$
- By adding a dot, we get a configuration (or an item)
  - $A \to \bullet X_1 X_2 \ldots X_j$
  - $A \to X_1 X_2 \ldots X_i \bullet X_{i+1} \ldots X_j$
  - $A \to X_1 X_2 \ldots X_j \bullet$
- The • indicates how much of a RHS has been shifted into the stack.

# LR(0) Parsing

- An item with the • at the end of the RHS
  - $A \rightarrow X_1 X_2 \ldots X_j \bullet$
  - indicates (or recognized) that RHS should be reduced to LHS
- An item with the • at the beginning of RHS
  - $A \rightarrow \bullet X_1 X_2 \ldots X_j$
  - predicts that RHS will be shifted into the stack

# LR(0) Parsing

- An LR(0) state is a set of configurations
  - This means that the actual state of LR(0) parsers is denoted by one of the items.
- The closure0 operation:
  - if there is an configuration $\mathbf{B \rightarrow \delta \bullet A\ \rho}$ in the set then add all configurations of the form $\mathbf{A \rightarrow \bullet\ \gamma}$ to the set.
- The initial configuration
  - $\mathbf{s0 = closure0(\{S \rightarrow \bullet\ \alpha\ \$\})}$

# LR(0) Parsing

```
configuration_set closure0(configuration_set s)
{
    configuration_set s′ = s;

    do {
        if (B →δ • Aρ ∈ s′ for A∈Vn) {
            /*
             * Predict productions with A
             * as the left-hand side.
             */
            Add all configurations of the form
                A → • γ to s′
        }
    } while (more new configurations can be added)

    return s′;
}
```

**Figure 6.5** An Algorithm to Close LR(0) Configuration Sets

$$S \rightarrow E\$$$
$$E \rightarrow E + T \mid T$$
$$T \rightarrow ID \mid (E)$$

$$\text{closure0}(\{S \rightarrow \bullet E\$\}) = \{\; S \rightarrow \bullet E\$,\; E \rightarrow \bullet E + T,\; E \rightarrow \bullet T,\; T \rightarrow \bullet ID,\; T \rightarrow \bullet (E) \;\}$$

# LR(0) Parsing

- Given a *configuration set* **s**, we can compute its successor, **s'**, under a symbol **X**
  - Denoted go_to0(s,X)=s'

```
configuration_set go_to0(configuration_set s, symbol X)
{
   s_b = ∅;
   for (each configuration c ∈ s)
      if (c is of the form A →β • X γ)
         Add A →βX • γ to s_b;

   /*
    * That is, we advance the • past the symbol X,
    * if possible.  Configurations not having a
    * dot preceding an X are not included in s_b.
    */

   /* Add new predictions to s_b via closure0. */
   return closure0(s_b);
}
```

**Figure 6.6** An Algorithm to Compute the LR(0) `go_to` Function

# LR(0) Parsing

- Characteristic finite state machine (CFSM)
  - It is a finite automaton, p.148, para. 2.
  - Identifying configuration sets and successor operation with CFSM states and transitions

```
void build_CFSM(void)
{
   Create the Start State of the CFSM; Label it with s0
   Create an Error State in the CFSM; Label it with ∅

   S = SET_OF( s0 );

   while(S is nonempty) {
      Remove a configuration set s from S;
      /* Consider both terminals and nonterminals */
      for (X in Symbols) {
         if (go_to0(s,X) does not label a CFSM state) {
            Create a new CFSM state and label it
               with go_to0(s,X);
            Put go_to0(s,X) into S;
         }
         Create a transition under X from the state s
            labels to the state go_to0(s,X) labels;
      }
   }
}
```

**Figure 6.7** An Algorithm to Compute the CFSM for a Grammar

# LR(0) Parsing

- For example, given grammar $G_2$

    $S' \rightarrow S\$$

    $S \rightarrow ID|\lambda$

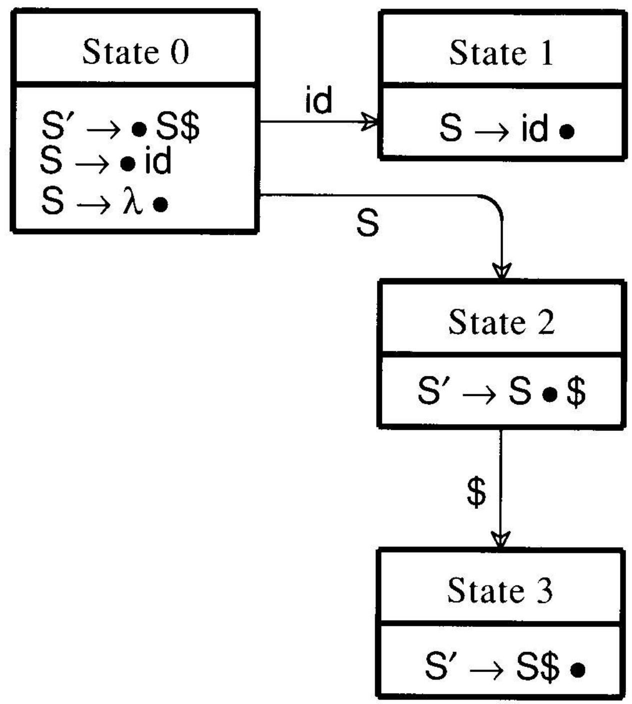



**Figure 6.8** CFSM for $G_2$

# LR(0) Parsing

- CFSM is the goto table of LR(0) parsers.

```
int ** build_go_to_table(finite_automaton CFSM)
{
   const int N = num_states(CFSM);
   int **tab;

   Dynamically allocate a table of dimension
      N × num_symbols(CFSM) to represent
      the go_to table and assign it to tab;

   Number the states of CFSM from 0 to N-1,
      with the Start State labeled 0;

   for (S = 0; S <= N - 1; S++) {
      /* Consider both terminals and nonterminals. */
      for (X in Symbols) {
         if (State S has a transition under X
             to some state T)
            tab[S][X] = T;
         else
            tab[S][X] = EMPTY;
      }
   }
   return tab;
}
```

**Figure 6.9** An Algorithm to Build the LR(0) `go_to` Table

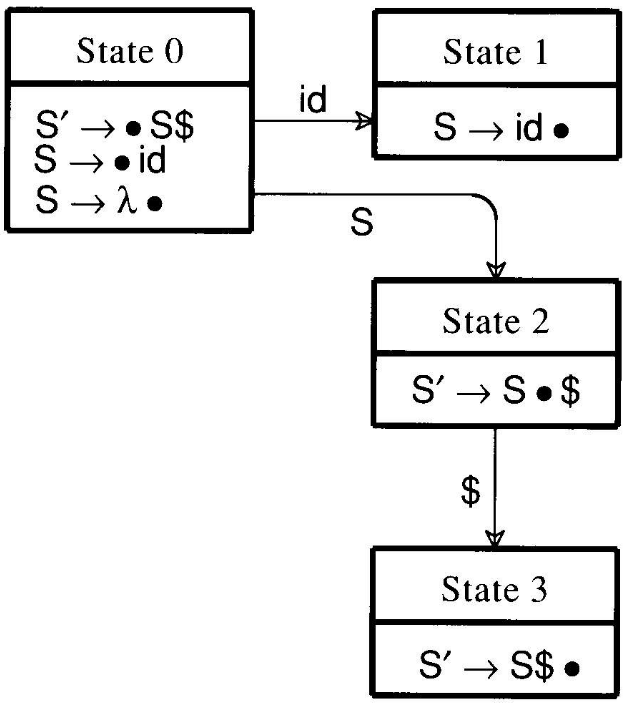



**Figure 6.8** CFSM for $G_2$

| State | Symbol | | |
|---|---|---|---|
| | ID | $ | S |
| 0 | 1 | 4 | 2 |
| 1 | 4 | 4 | 4 |
| 2 | 4 | 3 | 4 |
| 3 | 4 | 4 | 4 |
| 4 | | | |

**Figure 6.10** A `go_to` Table for Grammar $G_2$

# LR(0) Parsing

- Because LR(0) uses no lookahead, we must extract the action function directly from the configuration sets of CFSM
- Let $Q=\{Shift, Reduce_1, Reduce_2, \ldots, Reduce_n\}$
  - There are n productions in the CFG
- $S_0$ be the set of CFSM states
  - $P:S_0 \rightarrow 2^Q$
- $P(s)=\{Reduce_i \mid B \rightarrow \rho \bullet \in s$ and production i is $B \rightarrow \rho\} \cup$ (if $A \rightarrow \alpha \bullet a\beta \in s$ for a $\in V_t$ Then $\{Shift\}$ Else $\emptyset$)

# LR(0) Parsing

- G is LR(0) if and only if $\forall\ s \in S_0\ |P(s)|=1$
- If G is LR(0), the action table is trivially extracted from P
  - $P(s)=\{Shift\} \Rightarrow action[s]=Shift$
  - $P(s)=\{Reduce_j\}$, where production j is the augmenting production, $\Rightarrow action[s]=Accept$
  - $P(s)=\{Reduce_i\}$, $i \neq j$, $action[s]=Reduce_i$
  - $P(s)=\varnothing \Rightarrow action[s]=Error$

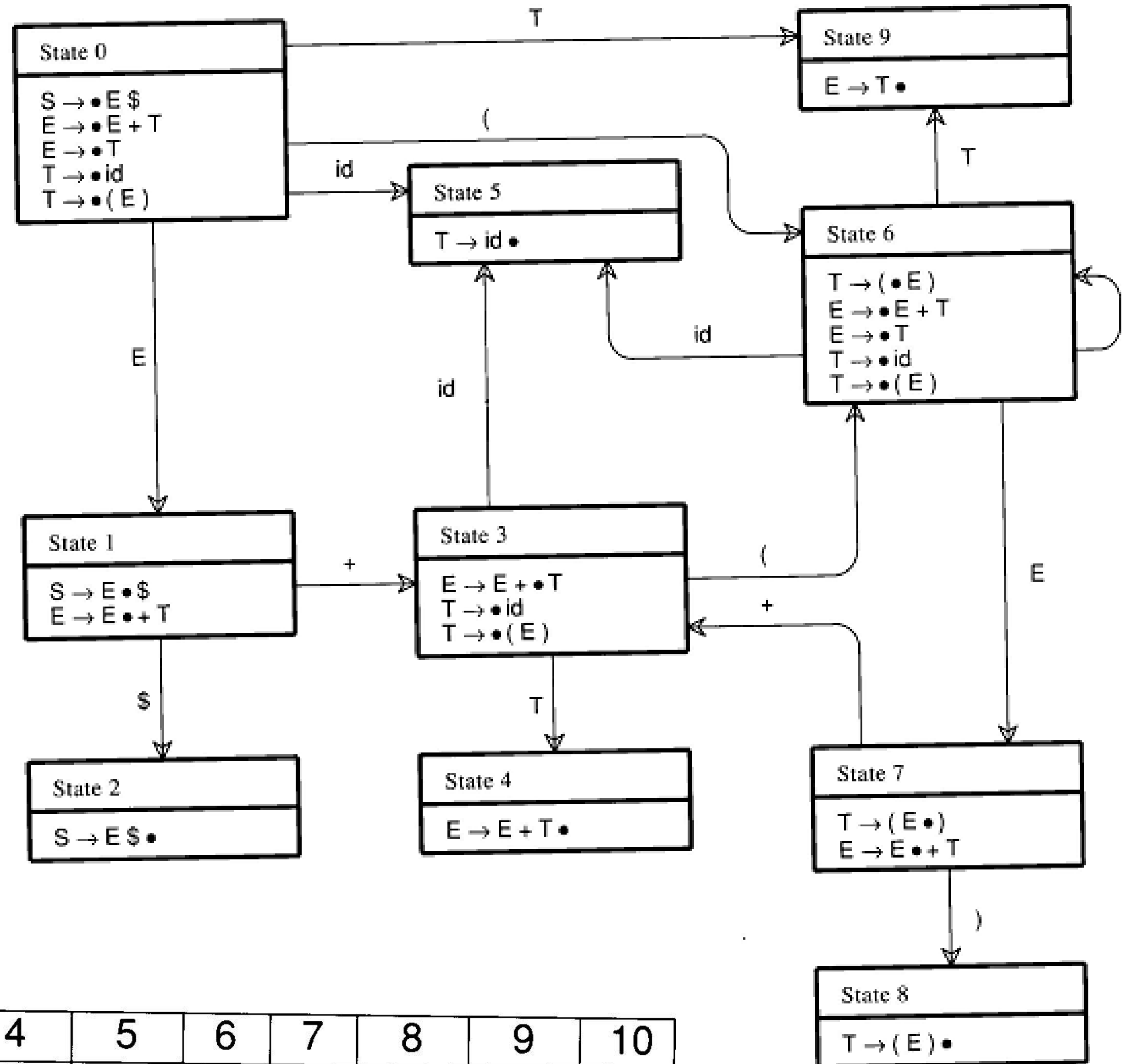
- Consider $G_1$

  $S \rightarrow E\$$

  $E \rightarrow E+T \mid T$

  $T \rightarrow ID \mid (E)$

**CFSM for $G_1 \Rightarrow$**




| State: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Action: | S | S | A | S | R2 | R4 | S | S | R5 | R3 | |

**Figure 6.12** `action Table for` $G_1$

# LR(0) Parsing

- Any state $s \in S_0$ for which $|P(s)|>1$ is said to be *inadequate*
- Two kinds of parser conflicts create inadequacies in configuration sets
  - Shift-reduce conflicts
  - Reduce-reduce conflicts

# LR(0) Parsing

- It is easy to introduce inadequacies in CFSM states
  - Hence, few real grammars are LR(0). For example,
    - Consider $\lambda$-productions
      - The only possible configuration involving a $\lambda$-production is of the form $A \rightarrow \lambda \bullet$
      - However, if A can generate any terminal string other than $\lambda$, then a shift action must also be possible (First(A))
    - LR(0) parser will have problems in handling operator precedence properly

# LR(1) Parsing

- An LR(1) configuration, or item is of the form
  - $A \rightarrow X_1 X_2 \ldots X_i \bullet X_{i+1} \ldots X_j$, ***l*** where ***l*** $\in V_t \cup \{\lambda\}$
    - The look ahead commponent ***l*** represents a possible lookahead after the entire right-hand side has been matched
    - The $\lambda$ appears as lookahead only for the augmenting production because there is no lookahead after the endmarker

# LR(1) Parsing

- We use the following notation to represent the set of LR(1) configurations that shared the same dotted production

$$A \rightarrow X_1 X_2 \ldots X_i \bullet X_{i+1} \ldots X_j,\ \{l_1 \ldots l_m\}$$

$$= \{A \rightarrow X_1 X_2 \ldots X_i \bullet X_{i+1} \ldots X_j,\ l_1\} \cup$$

$$\{A \rightarrow X_1 X_2 \ldots X_i \bullet X_{i+1} \ldots X_j,\ l_2\} \cup$$

$$\ldots$$

$$\{A \rightarrow X_1 X_2 \ldots X_i \bullet X_{i+1} \ldots X_j,\ l_m\}$$

# LR(1) Parsing

- There are many more distinct LR(1) configurations than LR(0) configurations.
- In fact, the major difficulty with LR(1) parsers is not their power but rather finding ways to represent them in storage-efficient ways.

# LR(1) Parsing

- Parsing begins with the configuration
  - **closure1({S→ • α $, {λ}})**

```
configuration_set closure1(configuration_set s)
{
   configuration_set s′ = s;

   do {
      if (B →δ • Aρ, l ∈s′ for A∈Vn) {
         /*
          * Predict productions with A as the
          * left-hand side. Possible lookaheads
          * are First(ρl)
          */
         Add all configurations of the form A → • γ, u,
            where  u ∈First(ρl), to s′
      }
   } while (more new configurations can be added)
   return s′;
}
```

**Figure 6.13** An Algorithm to Close LR(1) Configuration Sets

# LR(1) Parsing

- Consider $G_1$

  $S \rightarrow E\$$

  $E \rightarrow E+T \mid T$

  $T \rightarrow ID \mid (E)$

- closure1($S \rightarrow \bullet\, E\$$, $\{\lambda\}$)

---

**$S \rightarrow \bullet\, E\$, \{\lambda\}$**

**$E \rightarrow \bullet\, E+T, \{\$\}$**

**$E \rightarrow \bullet\, T, \{\$\}$**

**$E \rightarrow \bullet\, E+T, \{+\}$**

**$E \rightarrow \bullet\, T, \{+\}$**

**$T \rightarrow \bullet\, ID, \{\$\}$**

**$T \rightarrow \bullet\, (E), \{\$\}$**

**$T \rightarrow \bullet\, ID, \{+\}$**

**$T \rightarrow \bullet\, (E), \{+\}$**

closure1($S \rightarrow \bullet\, E\$$, $\{\lambda\}$)=

{

$S \rightarrow \bullet\, E\$, \{\lambda\}$;

$E \rightarrow \bullet\, E+T, \{\$+\}$

$E \rightarrow \bullet\, T, \{\$+\}$

$T \rightarrow \bullet\, ID, \{\$+\}$

$T \rightarrow \bullet\, (E), \{\$+\}$

}

How many configures?

# LR(1) Parsing

- Given an LR(1) configuration set s, we compute its successor, s', under a symbol X
  - go_to1(s,X)

```
configuration_set go_to1(configuration_set s, symbol X)
{
   sb = ∅;
   for (each configuration c ∈ s)
      if (c is of the form A →β • X γ, l)
         Add A →βX • γ, l to sb;

   /*
    * That is, we advance the • past the symbol X,
    * if possible.  Configurations not having a
    * dot preceding an X are not included in sb.
    */

   /* Add new predictions to sb via closure1. */
   return closure1(sb);
}
```

Figure 6.14 An Algorithm to Compute the LR(1) go_to Function

# LR(1) Parsing

- We can build a finite automation that is analogue of the LR(0) CFSM
  - LR(1) FSM, LR(1) machine
- The relationship between CFSM and LR(1) macine
  - By merging LR(1) machine’s configuration sets, we can obtain CFSM

```
void build_LR1(void)
{
 Create the Start State of the FSM; Label it with s0
 Put s0 into an initially empty set, S.
 while (S is nonempty) {
     Remove a configuration set s from S;
     /* Consider both terminals and nonterminals */
     for (X in Symbols) {
         if (go_to1(s,X) != ∅) {
             if (go_to1(s,X) does not label a FSM state) {
                 Create a new FSM state and label it
                     with go_to1(s,X);
                 Put go_to1(s,X) into S;
             }
             Create a transition under X from the
                 state s labels to the state
                 go_to1(s,X) labels;
         }
     }
 }
}
```

**Figure 6.15** An Algorithm to Build an LR(1) FSM

- $G_3$

  $S \rightarrow E\$$

  $E \rightarrow E+T \mid T$

  $T \rightarrow T*P \mid P$

  $P \rightarrow ID \mid (E)$

Is $G_3$ an LR(0) Grammar?

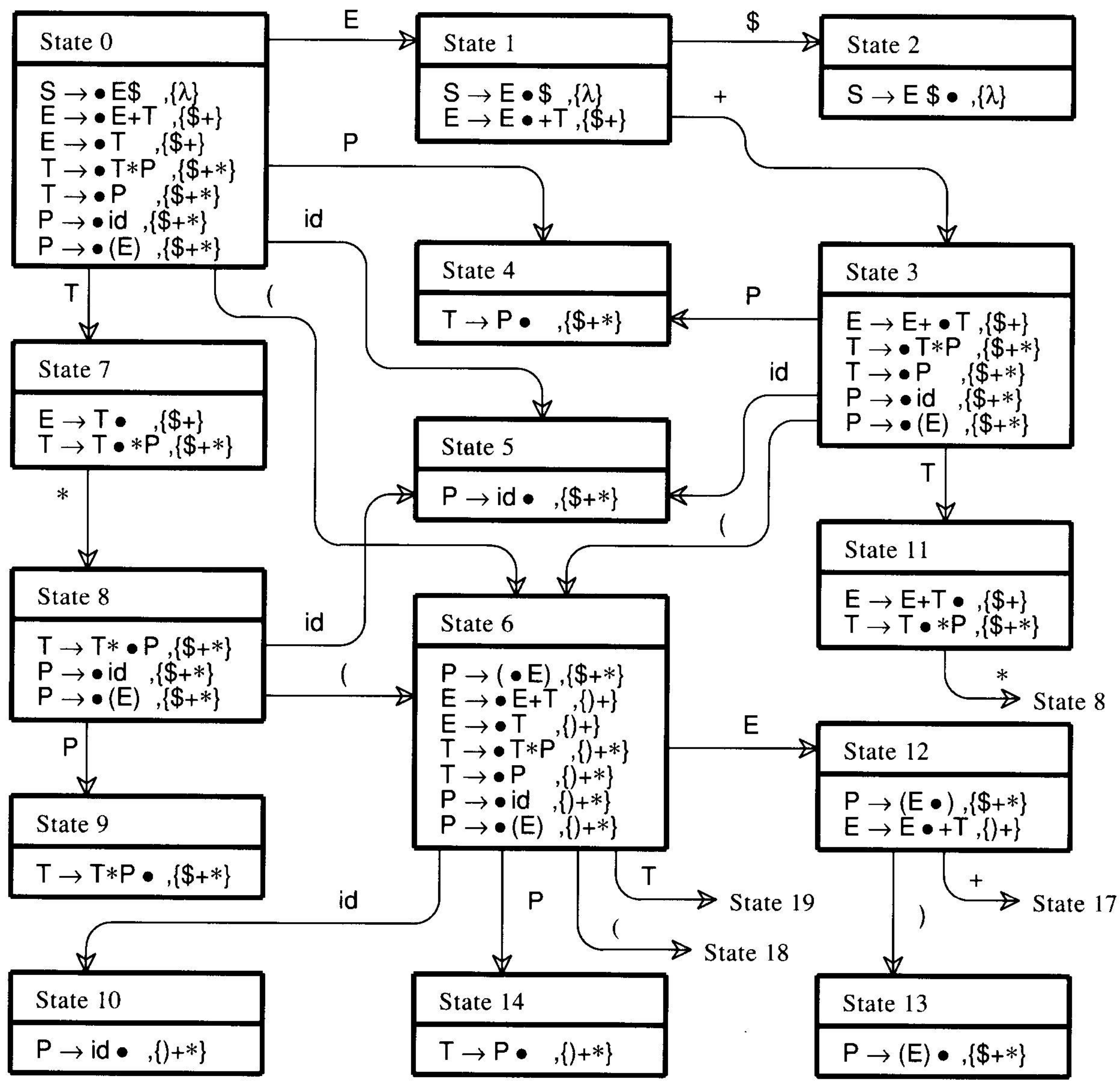



**Figure 6.16** LR(1) Machine for $G_3$

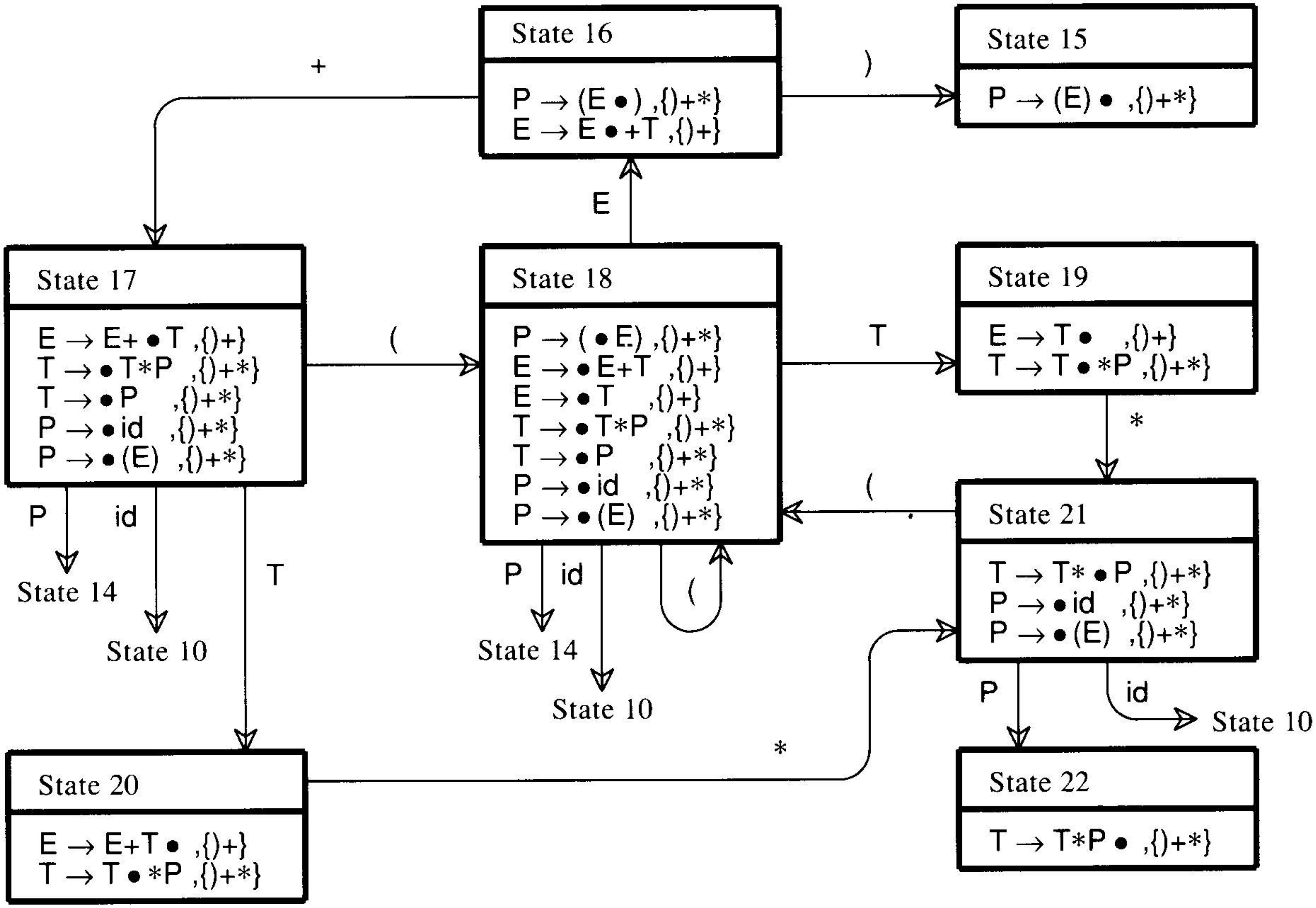



**Figure 6.16** (continued)

# LR(1) Parsing

- The go_to table used to drive an LR(1) is extracted directly from the LR(1) machine

```
int ** build_go_to_table(finite_automaton CFSM)
{
   const int N = num_states(CFSM);
   int **tab;

   Dynamically allocate a table of dimension
      N × num_symbols(CFSM) to represent
      the go_to table and assign it to tab;

   Number the states of CFSM from 0 to N-1,
      with the Start State labeled 0;

   for (S = 0; S <= N - 1; S++) {
      /* Consider both terminals and nonterminals. */
      for (X in Symbols) {
         if (State S has a transition under X
             to some state T)
            tab[S][X] = T;
         else
            tab[S][X] = EMPTY;
      }
   }
   return tab;
}
```

**Figure 6.9** An Algorithm to Build the LR(0) **go_to** Table

# LR(1) Parsing

- Action table is extracted directly from the configuration sets of the LR(1) machine
- A *projection function*, P
  - $P : S_1 \times V_t \rightarrow 2^Q$
    - $S_1$ be the set of LR(1) machine states
- $P(s,a)=\{Reduce_i \mid B \rightarrow \rho \bullet, a \in s$ and production i is $B \rightarrow \rho\} \cup$ (if $A \rightarrow \alpha \bullet a\beta, b \in s$ Then $\{Shift\}$ Else $\emptyset$)

# LR(1) Parsing

- G is LR(1) if and only if
  - $\forall s \in S_1 \ \forall a \in V_t \ \left|P(s,a)\right| \leq 1$
- If G is LR(1), the action table is trivially extracted from P
  - $P(s,\$)=\{Shift\} \Rightarrow action[s][\$]=Accept$
  - $P(s,a)=\{Shift\}, a \neq \$ \Rightarrow action[s][a]=Shift$
  - $P(s,a)=\{Reduce_i\}, \Rightarrow action[s][a]=Reduce_i$
  - $P(s,a)=\emptyset \ \Rightarrow action[s][a]=Error$

| State | Lookahead | | | | | |
|---|---|---|---|---|---|---|
| | + | * | ID | ( | ) | $ |
| 0 | | | S | S | | |
| 1 | S | | | | | A |
| 2 | | | | | | |
| 3 | | | S | S | | |
| 4 | R5 | R5 | | | | R5 |
| 5 | R6 | R6 | | | | R6 |
| 6 | | | S | S | | |
| 7 | R3 | S | | | | R3 |
| 8 | | | S | S | | |
| 9 | R4 | R4 | | | | R4 |
| 10 | R6 | R6 | | | R6 | |
| 11 | R2 | S | | | | R2 |
| 12 | S | | | | S | |
| 13 | R7 | R7 | | | | R7 |
| 14 | R5 | R5 | | | R5 | |
| 15 | R7 | R7 | | | R7 | |
| 16 | S | | | | S | |
| 17 | | | S | S | | |
| 18 | | | S | S | | |
| 19 | R3 | S | | | R3 | |
| 20 | R2 | S | | | R2 | |
| 21 | | | S | S | | |
| 22 | R4 | R4 | | | R4 | |



**Figure 6.17** LR(1) `action` Function for $G_3$

# SLR(1) Parsing

- LR(1) parsers are the most powerful class of shift-reduce parsers, using a single lookahead
  - *LR(1) grammars exist for virtually all programming languages*
  - LR(1)'s problem is that the LR(1) machine contains so many states that the go_to and action tables become prohibitively large

# SLR(1) Parsing

- In reaction to the space inefficiency of LR(1) tables, computer scientists have devised parsing techniques that are almost as powerful as LR(1) but that require far smaller tables
  1. One is to start with the CFSM, and then add lookahead after the CFSM is build
     - SLR(1)
  2. The other approach to reducing LR(1)'s space inefficiencies is to merger inessential LR(1) states
     - LALR(1)

# SLR(1) Parsing

- SLR(1) stands for *Simple LR(1)*
  - One-symbol lookahead
  - Lookaheads are not built directly into configurations but rather are added after the LR(0) configuration sets are built
  - An SLR(1) parser will perform a reduce action for configuration $\mathbf{B \rightarrow \rho \bullet}$ if the lookahead symbol is in the set **Follow(B)**

# SLR(1) Parsing

- The SLR(1) projection function, from CFSM states,
  - $P : S_0 \times V_t \rightarrow 2^Q$
  - $P(s,a)=\{Reduce_i \mid B \rightarrow \rho \bullet, a \in Follow(B)$ and production i is $B \rightarrow \rho \} \cup$ (if $A \rightarrow \alpha \bullet a\beta \in s$ for $a \in V_t$ Then $\{Shift\}$ Else $\emptyset$)

# SLR(1) Parsing

- G is SLR(1) if and only if
  - $\forall s \in S_0\ \forall a \in V_t\ |P(s,a)| \leq 1$
- If G is SLR(1), the action table is trivially extracted from P
  - $P(s,\$)=\{Shift\} \Rightarrow action[s][\$]=Accept$
  - $P(s,a)=\{Shift\}, a \neq \$ \Rightarrow action[s][a]=Shift$
  - $P(s,a)=\{Reduce_i\}, \Rightarrow action[s][a]=Reduce_i$
  - $P(s,a)=\varnothing \Rightarrow action[s][a]=Error$
- Clearly SLR(1) is a proper superset of LR(0)

# SLR(1) Parsing

- Consider $G_3$
  - It is LR(1) but not LR(0)
  - See states 7,11
  - Follow(E)={$,+,)}

$S \rightarrow E\$$

$E \rightarrow E+T|T$

$T \rightarrow T*P|P$

$P \rightarrow ID|(E)$

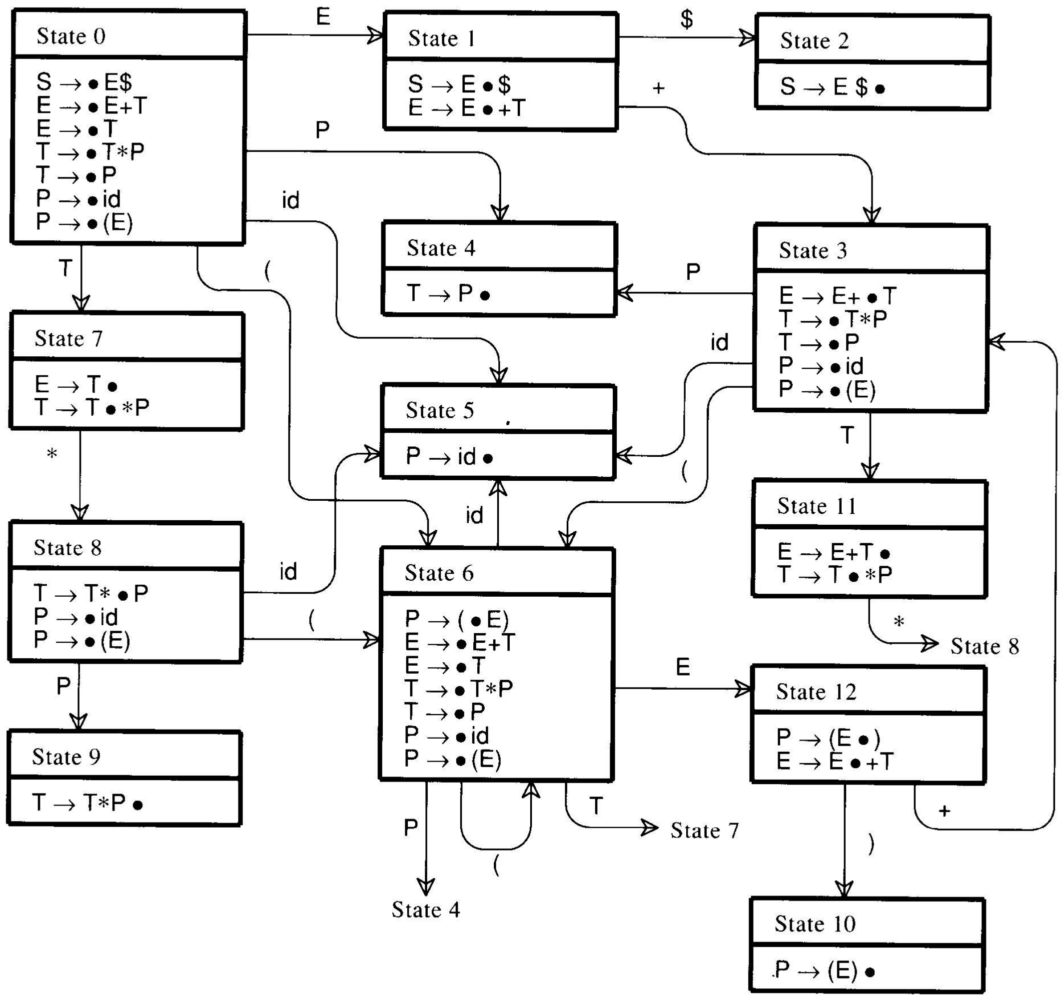



**Figure 6.18** CFSM for $G_3$

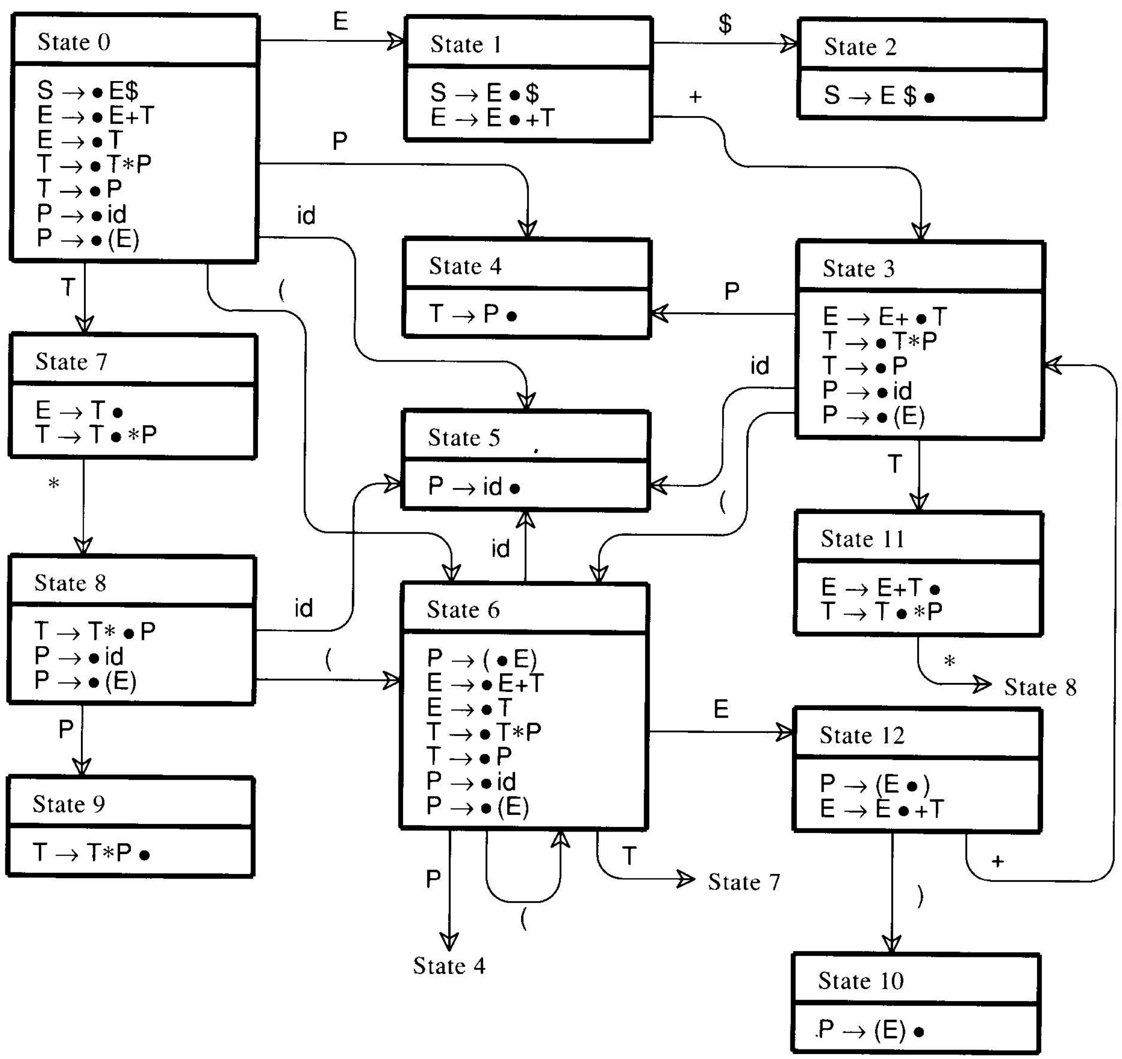



**Figure 6.18** CFSM for $G_3$

$S \rightarrow E\$$

$E \rightarrow E{+}T \mid T$

$T \rightarrow T{*}P \mid P$

$P \rightarrow ID \mid (E)$

| State | Lookahead | | | | | |
|---|---|---|---|---|---|---|
| | + | * | ID | ( | ) | $ |
| 0 | | | S | S | | |
| 1 | S | | | | | A |
| 2 | | | | | | |
| 3 | | | S | S | | |
| 4 | R5 | R5 | | | R5 | R5 |
| 5 | R6 | R6 | | | R6 | R6 |
| 6 | | | S | S | | |
| 7 | R3 | S | | | R3 | R3 |
| 8 | | | S | S | | |
| 9 | R4 | R4 | | | R4 | R4 |
| 10 | R7 | R7 | | | R7 | R7 |
| 11 | R2 | S | | | R2 | R2 |
| 12 | S | | | | S | |

**Figure 6.19** SLR(1) `action` Function for $G_3$

$G_3$ is both SLR(1) and LR(1).

# Limitations of the SLR(1) Technique

- The use of Follow sets to estimate the lookaheads that predict reduce actions is less precise than using the exact lookaheads incorporated into LR(1) configurations
  - Consider $G_4$

**Elem→(List, Elem)**

**Elem→Scalar**

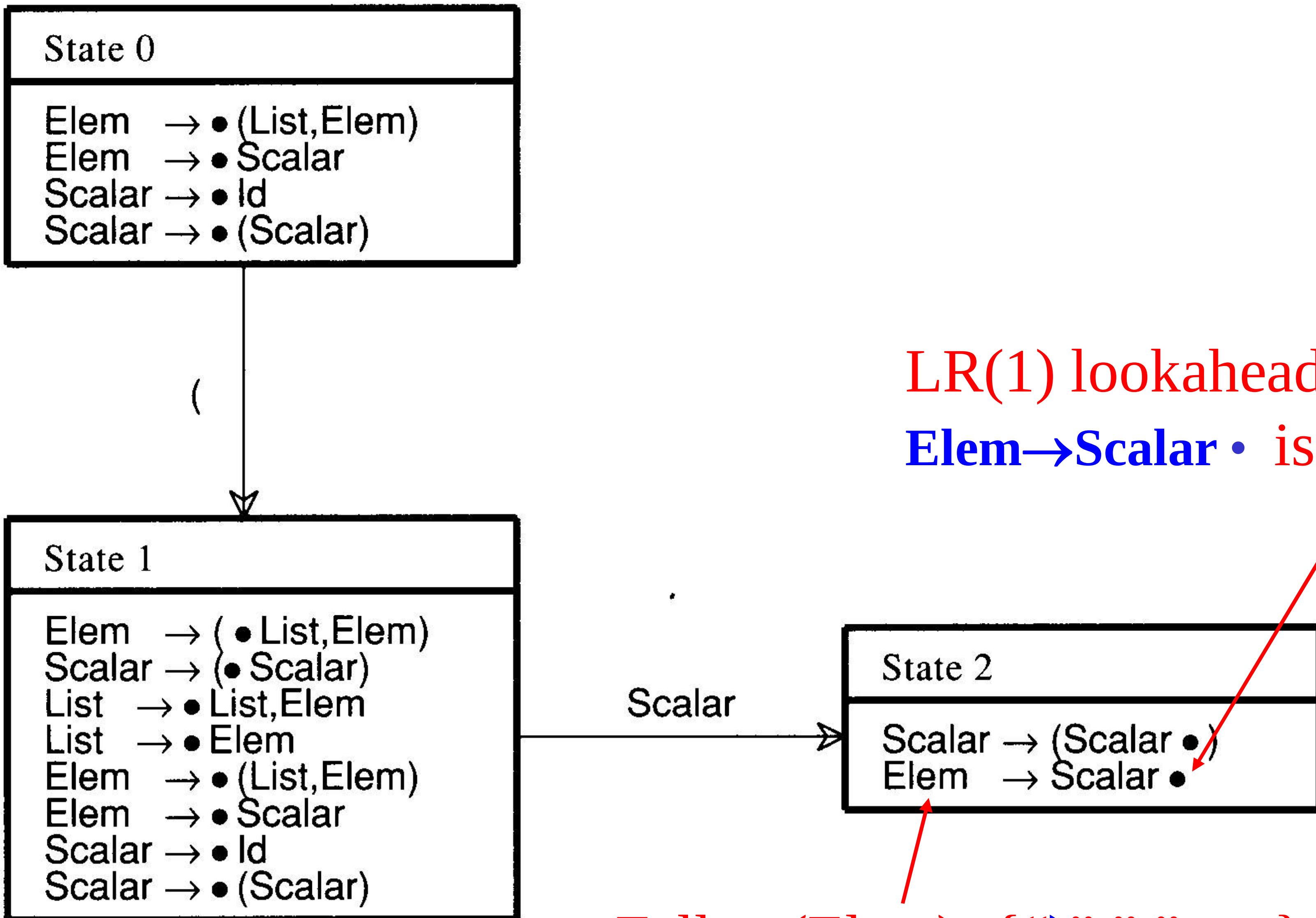
**List→List,Elem**

**List →Elem**

**Scalar →ID**

**Scalar→(Scalar)**

Fellow(Elem)={“**)**”,”**,**”,….}




LR(1) lookahead for

**Elem→Scalar •** is “,”

Fellow(Elem)={“)”,”,”,….}

**Figure 6.20** Part of the CFSM for $G_4$

# LALR(1)

- LALR(1) parsers can be built by first constructing an LR(1) parser and then merging states
  - An LALR(1) parser is an LR(1) parser in which all states that differ only in the lookahead components of the configurations are ***merged***
  - LALR is an acronym for ***L***ook ***A***head LR

# The core of a configuration

| State 3 |
| --- |
| E → E+ •T ,{$+} |
| T → •T*P ,{$+*} |
| T → •P ,{$+*} |
| P → •id ,{$+*} |
| P → •(E) ,{$+*} |

| State 17 |
| --- |
| E → E+ •T ,{)+} |
| T → •T*P ,{)+*} |
| T → •P ,{)+*} |
| P → •id ,{)+*} |
| P → •(E) ,{)+*} |

- The core of the above two configurations is the same

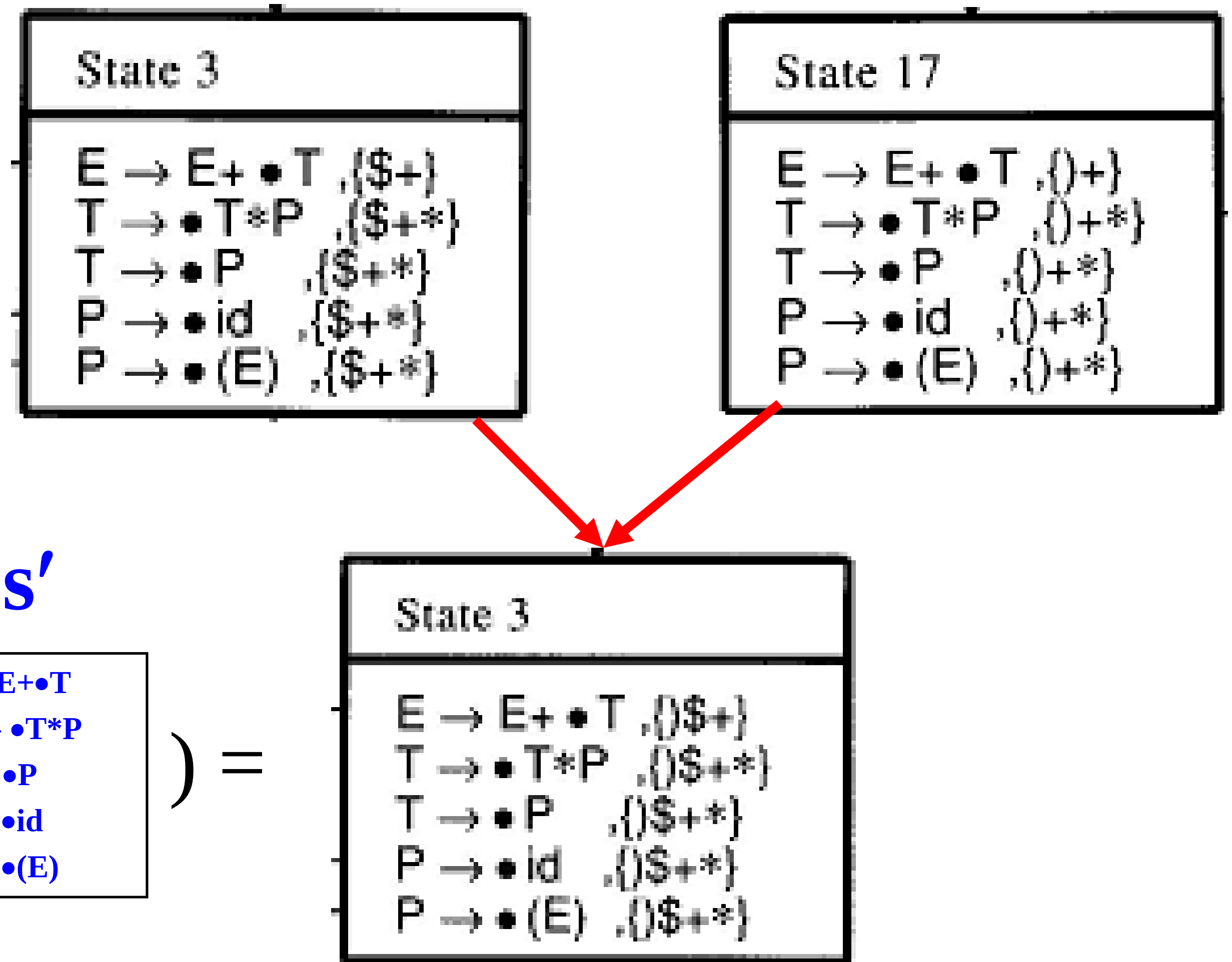
| |
| --- |
| **E→E+•T** |
| **T→ •T*P** |
| **T→ •P** |
| **P→ •id** |
| **P→ •(E)** |

# States Merge

- Cognate(**s′**)={c|c∈s, core(s)=**s′**}

**s′**

Cognate( E→E+•T / T→ •T*P / T→ •P / P→ •id / P→ •(E) ) =



# LALR(1)

- LALR(1) machine

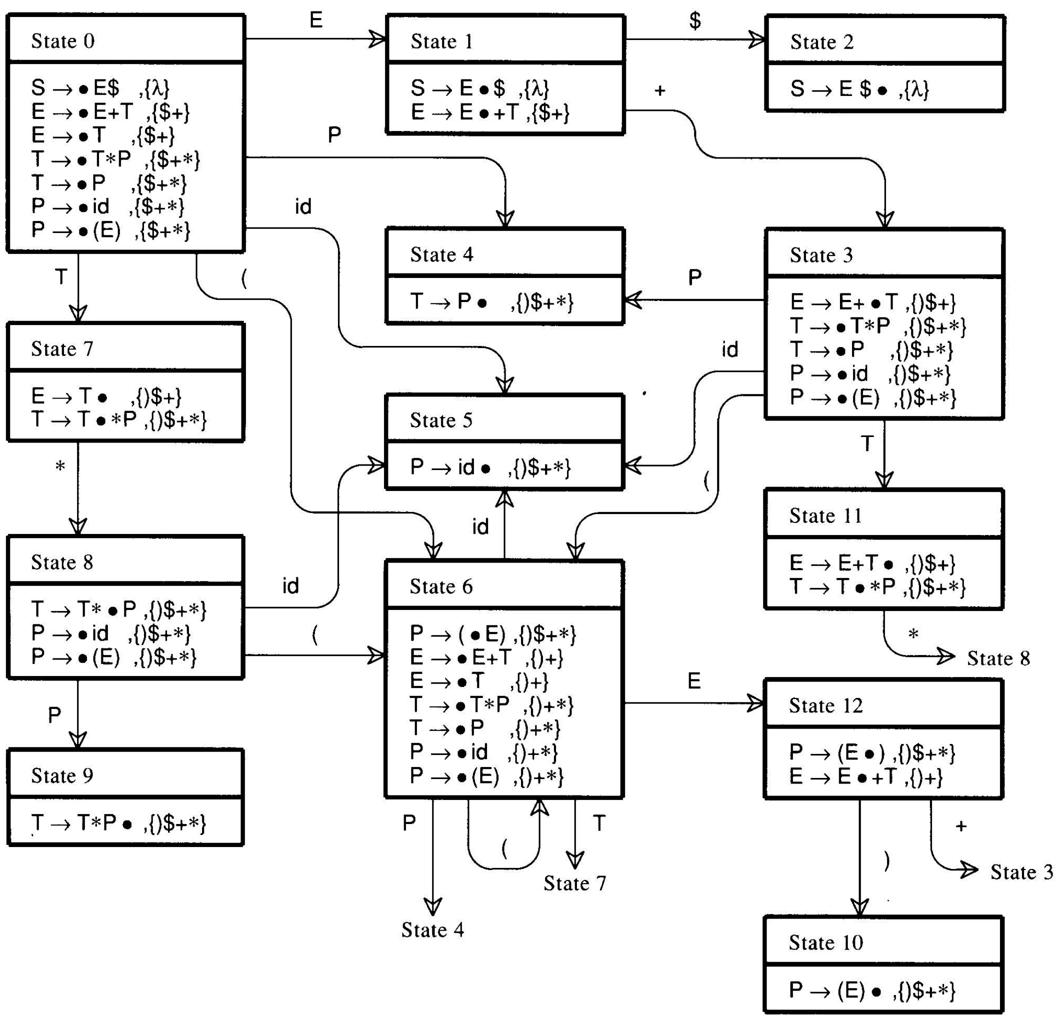



**Figure 6.22** LALR(1) Machine for $G_3$

| LALR(1) Cognate State | LR(1) States with Common Core |
|---|---|
| State 0 | State 0 |
| State 1 | State 1 |
| State 2 | State 2 |
| State 3 | State 3, State 17 |
| State 4 | State 4, State 14 |
| State 5 | State 5, State 10 |
| State 6 | State 6, State 18 |
| State 7 | State 7, State 19 |
| State 8 | State 8, State 21 |
| State 9 | State 9, State 22 |
| State 10 | State 13, State 15 |
| State 11 | State 11, State 20 |
| State 12 | State 12, State 16 |

**Figure 6.21** Cognate States for $G_3$

# LALR(1)

- The CFSM state is transformed into its LALR(1) Cognate
  - $P : S_0 \times V_t \rightarrow 2^Q$
  - $P(s,a)=\{Reduce_i \mid B\rightarrow\rho \bullet, a \in Cognate(s)$ and production i is $B\rightarrow\rho\ \} \cup$ (if $A\rightarrow \alpha\bullet a\beta \in s$ Then $\{Shift\}$ Else $\emptyset$)

# LALR(1) Parsing

- G is LALR(1) if and only if
  - $\forall s \in S_0 \; \forall a \in V_t \; \left|P(s,a)\right| \leq 1$
- If G is LALR(1), the action table is trivially extracted from P
  - $P(s,\$)=\{\text{Shift}\} \Rightarrow \text{action}[s][\$]=\text{Accept}$
  - $P(s,a)=\{\text{Shift}\}, a\neq\$ \Rightarrow \text{action}[s][a]=\text{Shift}$
  - $P(s,a)=\{\text{Reduce}_i\}, \Rightarrow \text{action}[s][a]=\text{Reduce}_i$
  - $P(s,a)=\emptyset \Rightarrow \text{action}[s][a]=\text{Error}$

# LALR(1) Parsing

- Consider $G_5$

  **<stmt>→ID**

  **<stmt>→<var>:=<expr>**

  **<var> →ID**
  **<var> →ID [<expr>]**
  **<expr>→<var>**

- Assume statements are separated by **;**'s, the grammar is not SLR(1) because

  **;** ∈ Follow(<stmt>) and

  **;** ∈ Follow(<var>), since <expr>→<var>

# LALR(1) Parsing

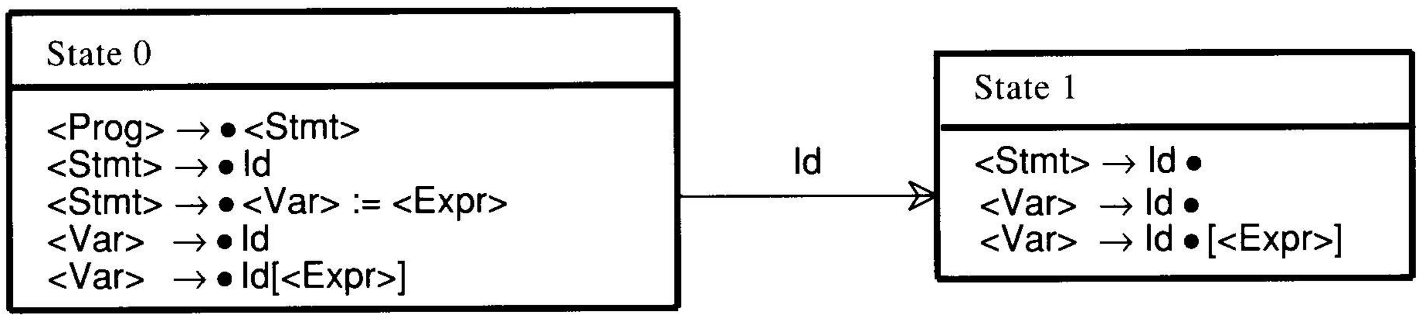



**Figure 6.23** Part of the CFSM for $G_5$

- However, in LALR(1), if we use <var> → ID the next symbol must be :=

  so action[ 1, := ] = reduce(<var> → ID)

  action[ 1, ; ] = reduce(<stmt> → ID)

  action[ 1,[ ] = shift

- There is no conflict.

# LALR(1) Parsing

- A common technique to put an LALR(1) grammar into SLR(1) form is to introduce a new nonterminal whose global (i.e. SLR) lookaheads more nearly correspond to LALR's exact look aheads
  - Follow(<lhs>) = {:=}

**<stmt>→ID**
**<stmt>→<var>:=<expr>**
**<var> →ID**
**<var> →ID [<expr>]**
**<expr>→<var>**

**<stmt>→ID**
**<stmt>→<lhs>:=<expr>**
**<lhs> →ID**
**<lhs> →ID [expr>]**
**<var> →ID**
**<var> →ID [expr>]**
**<expr>→<var>**

# LALR(1) Parsing

- At times, it is the CFSM itself that is at fault.

  **S→(Exp1)**

  **S→[Exp1]**

  **S→(Exp2]**
  **S→[Exp2)**
  **<Exp1>→ID**
  **<Exp2>→ID**

- A different expression nonterminal is used to allow error or warning diagnostics

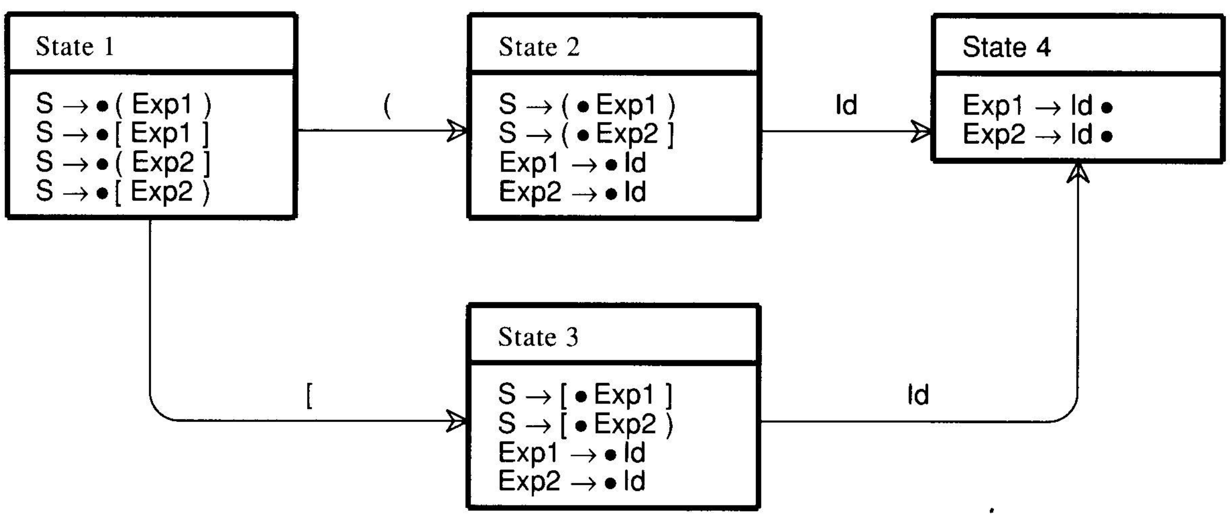



**Figure 6.24** Part of the CFSM for Grammar $G_6$

# Building LALR(1) Parsers

- In the definition of LALR(1)
  - An LR(1) machine is first built, and then its states are merged to form an automaton identical in structure to the CFSM
    - May be quite inefficient
  - An alternative is to build the CFSM first.
    - Then LALR(1) lookaheads are "***propagated***" from configuration to configuration

# Building LALR(1) Parsers

- Propagate links:
  - Case 1: one configuration is created from another in a previous state via a shift operation

$$\boxed{A \to \alpha \bullet X\gamma \, , \, L_1} \longrightarrow \boxed{A \to \alpha X \bullet \gamma \, , \, L_2}$$

# Building LALR(1) Parsers

- Propagate links:
  - Case 2: one configuration is created as the result of a closure or prediction operation on another configuration

$$
\begin{array}{l}
B \rightarrow \beta \bullet A \gamma \text{ , } L_1 \\
\quad\quad\downarrow \\
A \rightarrow \bullet \alpha \text{ , } L_2
\end{array}
$$

$$L_2 = \{\ x \mid x \in \text{First}(\gamma t) \text{ and } t \in L_1\ \}$$

# Building LALR(1) Parsers

- **Step 1**: After the CFSM is built, we can create all the necessary propagate links to transmit lookaheads from one configuration to another
- **Step 2**: spontaneous lookaheads are determined
  - By including in $L_2$, for configuration $A \rightarrow \bullet\alpha, L_2$, all spontaneous lookaheads induced by configurations of the form $B \rightarrow \beta \bullet A\gamma, L_1$
    - These are simply the non-$\lambda$ values of First($\gamma$)
- **Step 3**: Then, propagate lookaheads via the propagate links
  - See figure 6.25

# Building LALR(1) Parsers

```
while (stack is not empty)
{
   pop top item, assign its components to (s,c,L)

   if (configuration c in state s
          has any propagate links) {
      Try, in turn, to add L to the lookahead set of
         each configuration so linked.
      for (each configuration c̄ in state s̄
           to which L is added)
         Push (s̄,c̄,L) onto the stack.
   }
}
```

**Figure 6.25** LALR(1) Lookahead Propagation Algorithm

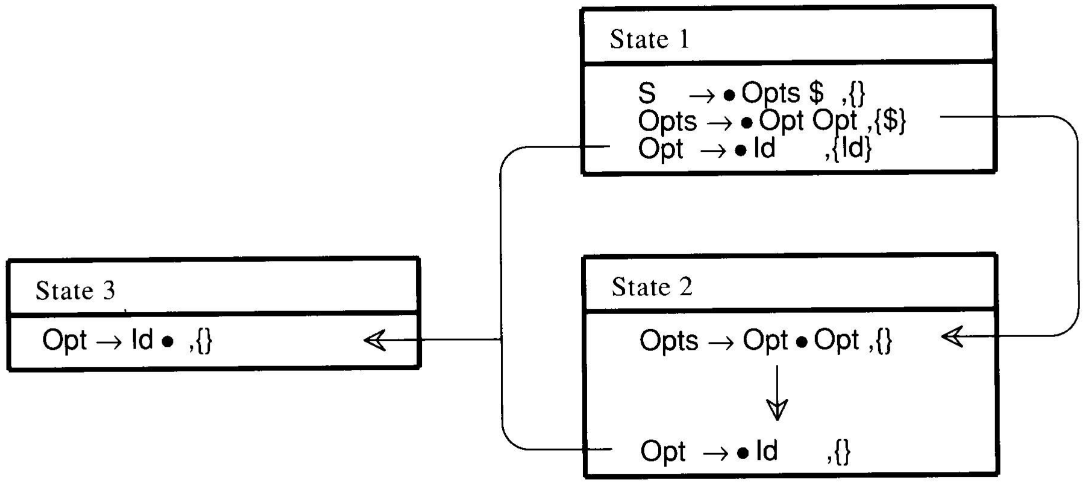



**Figure 6.26** Part of CFSM for $G_7$ with Propagate Links

| Step | Stack | Action |
|---|---|---|
| (1) | (s1,c2,$), (s1,c3,ID) | Pop (s1,c2,$)<br>Add $ to c1 in s2<br>Push (s2,c1,$) |
| (2) | (s2,c1,$), (s1,c3,ID) | Pop (s2,c1,$)<br>Add $ to c2 in s2<br>Push (s2,c2,$) |
| (3) | (s2,c2,$), (s1,c3,ID) | Pop (s2,c2,$)<br>Add $ to c1 in s3<br>Push (s3,c1,$) |
| (4) | (s3,c1,$), (s1,c3,ID) | Pop (s3,c1,$)<br>Nothing is added (no links) |
| (5) | (s1,c3,ID) | Pop (s1,c3,ID)<br>Add ID to c1 in s3<br>Push (s3,c1,ID) |
| (6) | (s3,c1,ID) | Pop (s3,c1,ID)<br>Nothing is added (no links) |
| (7) | Empty | Terminate algorithm |

**Figure 6.27** Example of Lookahead Propagation

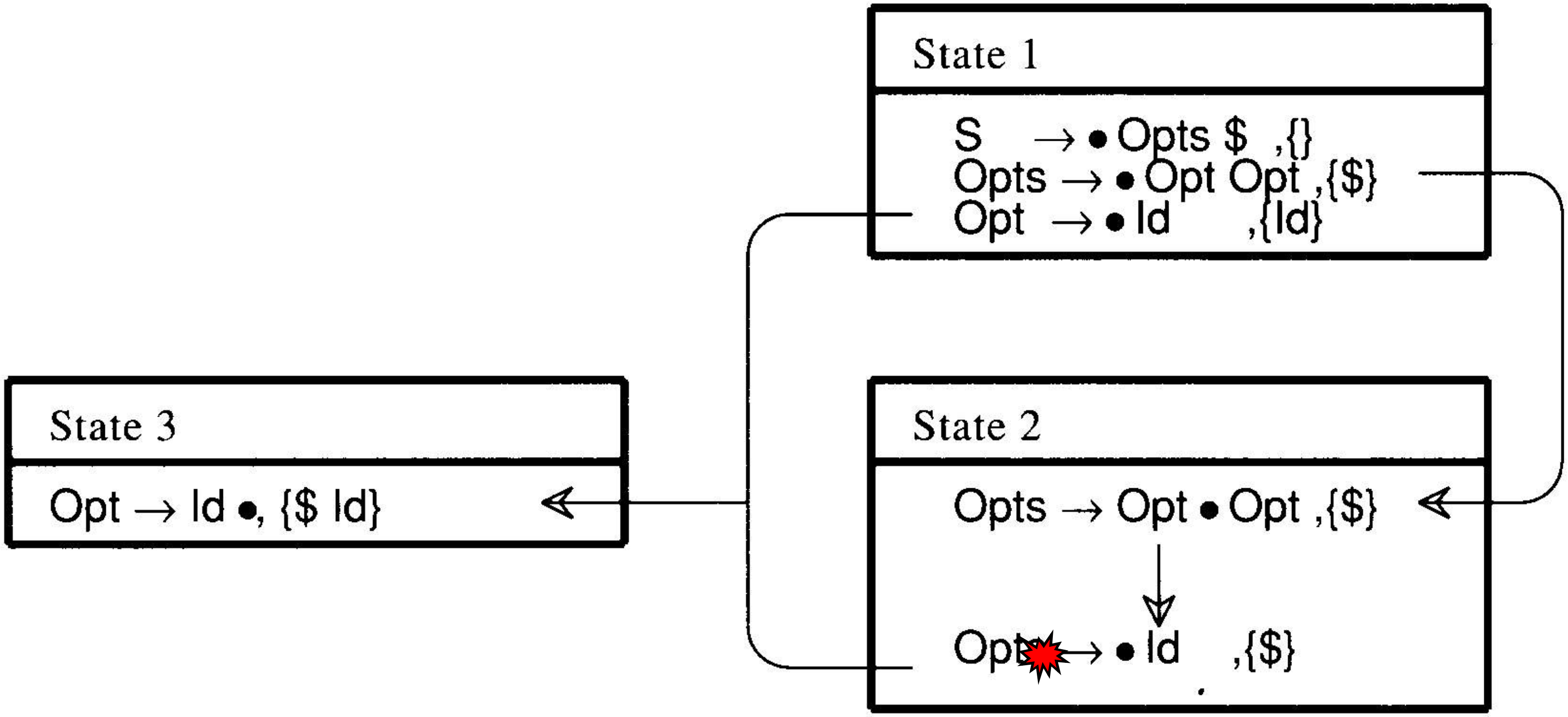



**Figure 6.28** Part of CFSM for $G_7$ with Lookaheads Propagated

# Building LALR(1) Parsers

- A number of LALR(1) parser generators use lookahead propagation to compute the parser action table
  - LALRGen uses the propagation algorithm
  - YACC examines each state repeatedly
- An intriguing alternative to propagating LALR lookaheads is to compute them as needed by doing a backward search through the CFSM
  - Read it yourself.  P. 176, Para. 3

# Building LALR(1) Parsers

- An intriguing alternative to propagating LALR lookaheads is to compute them as needed by doing a backward search through the CFSM
  - Read it yourself. P. 176, Para. 3

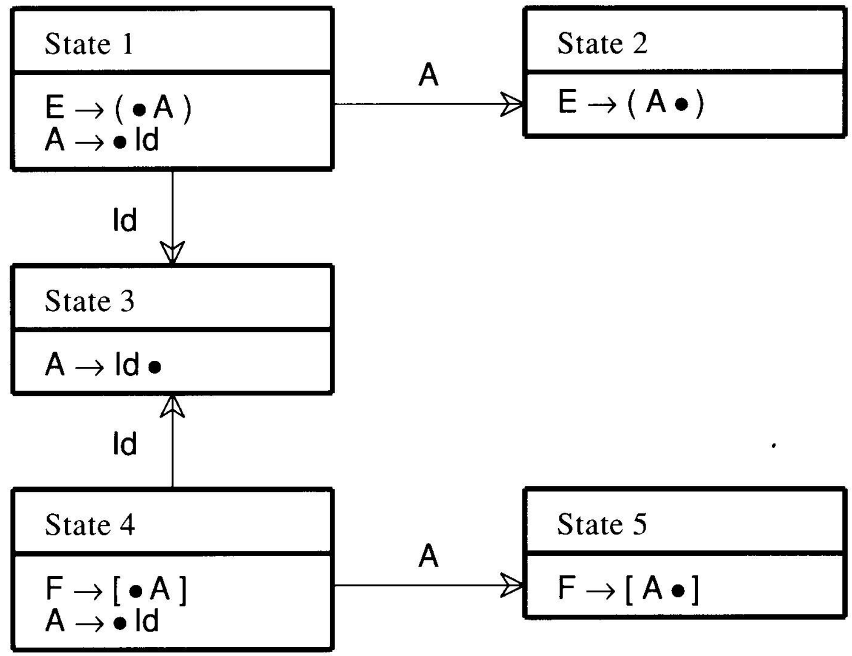



**Figure 6.29** A CFSM Analyzed Using Backward Search

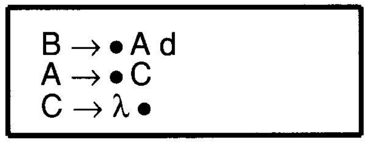



**Figure 6.30** A CFSM State that May Cause Backward Analysis to Fail

# Calling Semantic Routines in Shift-Reduce Parsers

- Shift-reduce parsers can normally handle larger classes of grammars than LL(1) parsers, which is a major reason for their popularity
- Shift-reduce parsers are not predictive, so we cannot always be sure what production is being recognized until its entire right-hand side has been matched
  - *The semantic routines can be invoked only after a production is recognized and reduced*
    - Action symbols only at the extreme right end of a right-hand side

# Calling Semantic Routines in Shift-Reduce Parsers

- Two common tricks are known that allow more flexible placement of semantic routine calls
- For example,

  <stmt>$\rightarrow$**if** <expr> **then** <stmts> **else** <stmts> **end if**

- We need to call semantic routines after the conditional expression *else* and *end if* are matched
  - Solution: create new nonterminals that generate $\lambda$

    <stmt>$\rightarrow$**if** <expr> <test cond>

    **then** <stmts> <process then part>

    **else** <stmts> **end if**

    <test cond>$\rightarrow\lambda$

    <process than part>$\rightarrow\lambda$

# Calling Semantic Routines in Shift-Reduce Parsers

- If the right-hand sides differ in the semantic routines that are to be called, the parser will be unable to correctly determine which routines to invoke
  - Ambiguity will manifest. For example,

<stmt>→**if** <expr> <test cond1>
**then** <stmts> <process then part>
**else** <stmts> **end if;**
<stmt>→**if** <expr> <test cond2>
**then** <stmts> <process then part>
**end if;**
<test cond1>→λ
<test cond2>→λ
<process than part>→λ

# Calling Semantic Routines in Shift-Reduce Parsers

- An alternative to the use of λ–generating nonterminals is to break a production into a number of pieces, with the breaks placed where semantic routines are required

  <stmt>→<if head><then part><else part>

  <if head>→**if** <expr>

  <then part>→**then** <stmts>

  <else part>→**then** <stmts> **end if;**

  – This approach can make productions harder to read but has the advantage that no λ–generating are needed

# Uses (and Misuses) of Controlled Ambiguity

- Research has shown that ambiguity, *if controlled*, can be of value in producing efficient parsers for real programming languages.
- The following grammar is not LR(1)

  Stmt$\rightarrow$ **if** Expr **then** Stmt

  Stmt$\rightarrow$ **if** Expr **then** Stmt **else** Stmt

# Uses (and Misuses) of Controlled Ambiguity

- The following grammar is not ambiguous

<stmt>→<matched>|<unmatched>

<matched>→ if <logic_expr> then <matched> else <matched>

| <any_non-if_statement>

<unmatched>→ if <logic_expr> then <stmt>

| if <logic_expr> then <matched> else <unmatched>

# Uses (and Misuses) of Controlled Ambiguity

- In LALRGen, when conflicts occur, we may use the option resolve to give preference to earlier productions.
- In YACC, conflicts are solved in
  - 1. shift is preferred to reduce;
  - 2. earlier productions are preferred.

- Grammars with conflicts are usually smaller.
- Application: operator precedence and associativity.

# Uses (and Misuses) of Controlled Ambiguity

- We no longer specify a parser purely in terms of the grammar it parses, but rather we explicitly include auxiliary rules to disambiguate conflicts in the grammar.
  - We will present how to specify auxiliary rules in yacc when we introduce the yacc.

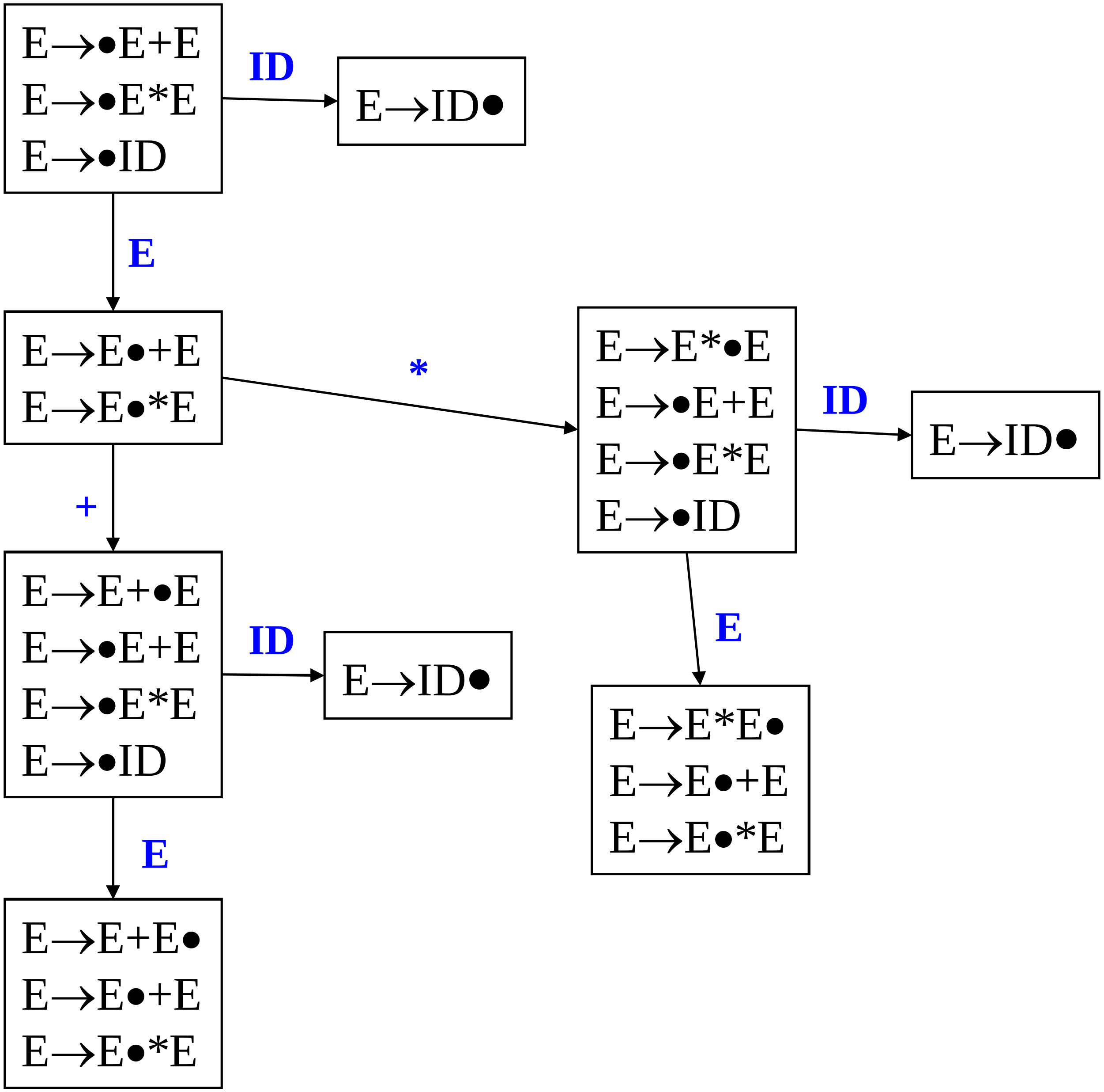
# Uses (and Misuses) of Controlled Ambiguity

- The precedence values assigned to operators resolve most shift-reduce conflicts of the form

  $\text{Expr} \rightarrow \text{Expr OP}_1 \text{ Expr} \bullet$

  $\text{Expr} \rightarrow \text{Expr} \bullet \text{ OP}_2 \text{ Expr}$

- Rules:
  - If Op1 has a higher precedence than Op2, we Reduce.
  - If Op2 has a higher precedence, we shift.
  - If Op1 and Op2 have the same precedence, we use the associativity definitions.
    - Right-associative, we shift. Left-associative, we reduce.
    - No-associative, we signal, and error.


E→●E+E
E→●E*E
E→●ID
ID
E→ID●
E
E→E●+E
E→E●*E
*
E→E*●E
E→●E+E
E→●E*E
E→●ID
ID
E→ID●
+
E→E+●E
E→●E+E
E→●E*E
E→●ID
ID
E→ID●
E
E→E*E●
E→E●+E
E→E●*E
E
E→E+E●
E→E●+E
E→E●*E

# Uses (and Misuses) of Controlled Ambiguity (Cont'd)

- More example: Page 234 of "lex&yacc"

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE
%%

stmt:   IF expr stmt        %prec LOWER_THAN_ELSE
      | IF expr stmt ELSE stmt;
```

# Uses (and Misuses) of Controlled Ambiguity (Cont'd)

- If your language uses a THEN keyword (like Pascal does):

```
%nonassoc THAN
%nonassoc ELSE
%%

stmt:   IF expr THAN stmt
      | IF expr THAN stmt ELSE stmt
      ;
```

# Optimizing Parse Tables

- A number of improvements can be made to decrease the space needs of an LR parse
  - Merging go_to and action tables to a single table

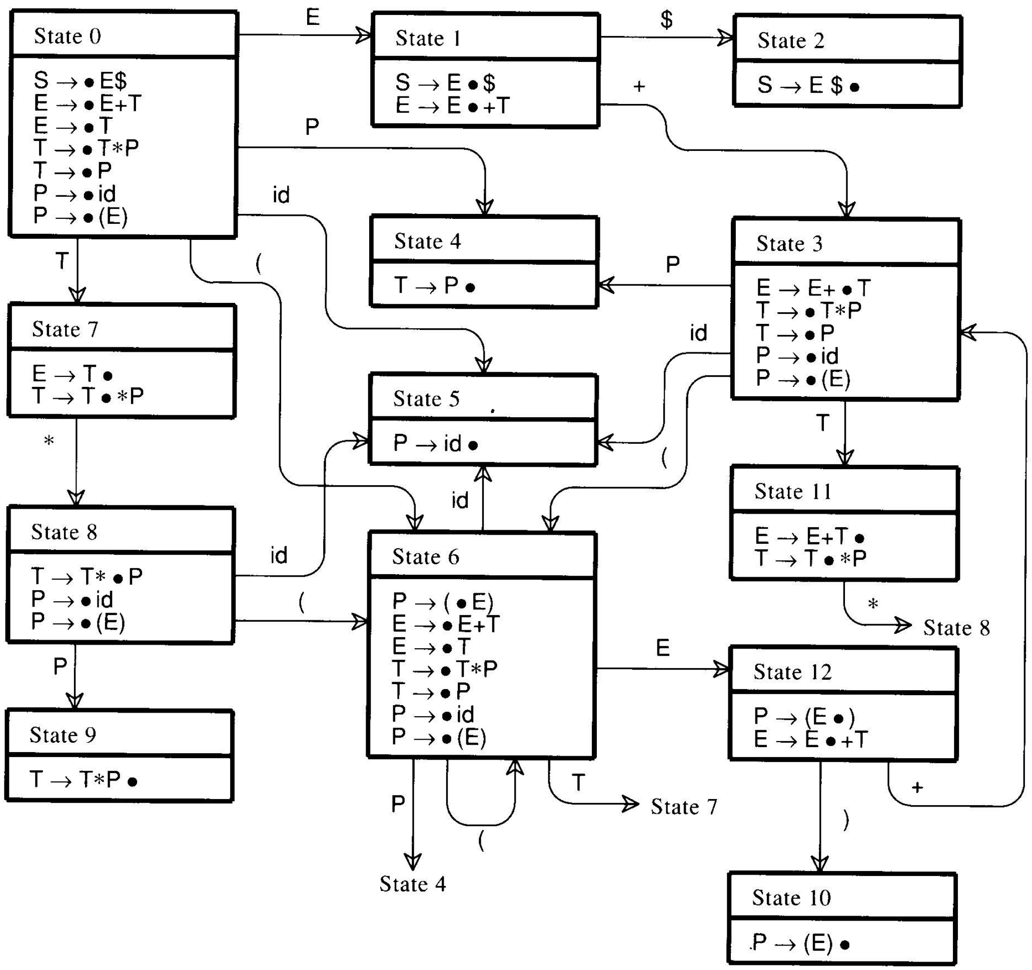



**Figure 6.18** CFSM for $G_3$

| State | Lookahead | | | | | |
|---|---|---|---|---|---|---|
| | + | * | ID | ( | ) | $ |
| 0 | | | S | S | | |
| 1 | S | | | | | A |
| 2 | | | | | | |
| 3 | | | S | S | | |
| 4 | R5 | R5 | | | R5 | R5 |
| 5 | R6 | R6 | | | R6 | R6 |
| 6 | | | S | S | | |
| 7 | R3 | S | | | R3 | R3 |
| 8 | | | S | S | | |
| 9 | R4 | R4 | | | R4 | R4 |
| 10 | R7 | R7 | | | R7 | R7 |
| 11 | R2 | S | | | R2 | R2 |
| 12 | S | | | | S | |

**Figure 6.19** SLR(1) `action` Function for $G_3$

| State | Symbol | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | + | * | ID | ( | ) | $ | E | T | P |
| 0 | | | S5 | S6 | | | S1 | S7 | S4 |
| 1 | S3 | | | | | A | | | |
| 2 | | | | | | | | | |
| 3 | | | S5 | S6 | | | | S11 | S4 |
| 4 | R5 | R5 | | | R5 | R5 | | | |
| 5 | R6 | R6 | | | R6 | R6 | | | |
| 6 | | | S5 | S6 | | | S12 | S7 | S4 |
| 7 | R3 | S8 | | | R3 | R3 | | | |
| 8 | | | S5 | S6 | | | | | S9 |
| 9 | R4 | R4 | | | R4 | R4 | | | |
| 10 | R7 | R7 | | | R7 | R7 | | | |
| 11 | R2 | S8 | | | R2 | R2 | | | |
| 12 | S3 | | | | S10 | | | | |

**Figure 6.35** SLR(1) Parse Table for $G_3$

# Optimizing Parse Tables

- Encoding parse table
  - As integers
    - Error entries as zeros
    - Reduce actions as positive integers
    - Shift actions as negative integers
- Single reduce states
  - E.g. state 4 in Figure 6.35
    - State 5 can be eliminated

# Optimizing Parse Tables

- See figure 6.36

| State | Symbol | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | + | * | ID | ( | ) | $ | E | T | P |
| 0 | | | L6 | S6 | | | S1 | S7 | L5 |
| 1 | S3 | | | | | A | | | |
| 3 | | | L6 | S6 | | | | S11 | L5 |
| 6 | | | L6 | S6 | | | S12 | S7 | L5 |
| 7 | R3 | S8 | | | R3 | R3 | | | |
| 8 | | | L6 | S6 | | | | | L4 |
| 11 | R2 | S8 | | | R2 | R2 | | | |
| 12 | S3 | | | | L7 | | | | |

**Figure 6.36** Optimized SLR(1) Parse Table for $G_3$