Compiler Techniques

3. Syntax Analysis

Huang Shell Ying

Overview

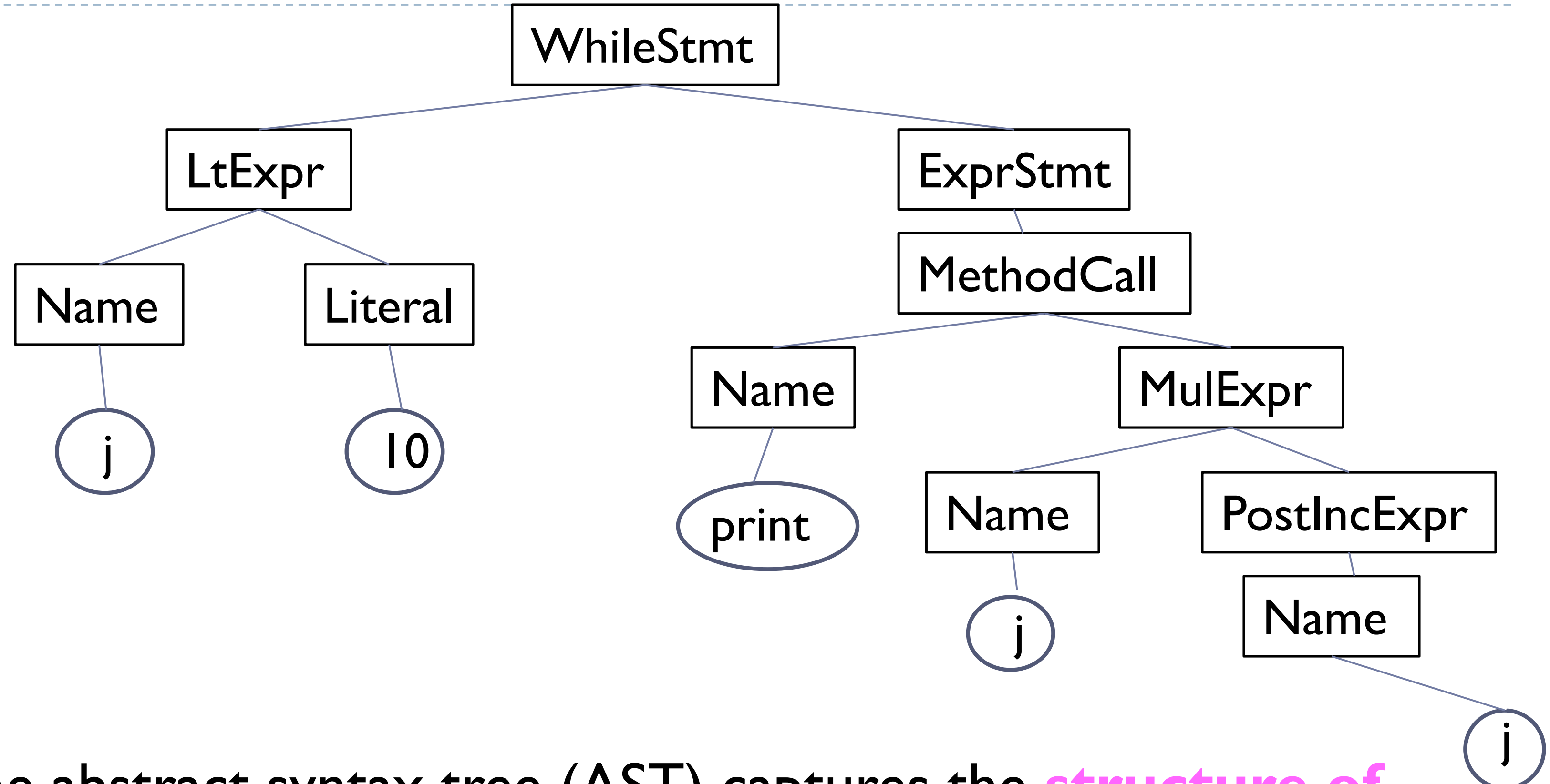
- A syntax analyser (parser) checks that the program is **syntactically well-formed** and transforms it from a sequence of tokens into an ***abstract syntax tree (AST)***.

For example,

```
while ( j < 10 ) print ( j * ( j ++ ) ) ;
```

WHILE	LPAREN	ID	LT	LIT	RPAREN	ID	LPAREN	ID	MUL	LPAREN	…

AST for while (j < 10) print (j * (j ++)) ;



- The abstract syntax tree (AST) captures the **structure of the program.**
- Later stages of the compiler use the AST for **semantic analysis** and **code generation**.

Overview

- A language is potentially an infinite set of sentences.
- To check whether a program is well-formed, we need a tool: The grammar of a language
- A language's grammar serves as a concise definition of how well-formed sentences in a language can be constructed.
- In this chapter, we will learn
  - Context-free grammar
  - The parsing problem
    - Top-down parsing
    - Bottom-up parsing

Context-Free Grammars

- Programming language constructs are recursive

  E.g.

  A **Stmt** is

  if Expr **Stmt** else **Stmt**

- C**ontext-free grammar** is able to define recursive structures (can we use regular expressions?).

Context-Free Grammars

- Programming language syntax can be described by a **context-free grammar**.
- A context-free grammar $G = (T, N, P, S)$ consists of four components:
  1. A finite set T of **terminals** (token types)
  2. A finite set N of **nonterminals** such that $T \cap N = \emptyset$
  3. A **start symbol** $S \in N$
  4. A finite set P of **rules** of the form $A \rightarrow s_1 \ldots s_n$ where $A \in N$, $n \geq 0$, and $\forall i \in \{1, \ldots, n\}$, $s_i \in T \cup N$.
     If $n = 0$, we write the rule as $A \rightarrow \lambda$.

Examples of Context-Free Grammar Rules

IFStat $\rightarrow$ if Cond then AssignStat OptionalElse

OptionalElse $\rightarrow$ else AssignStat

OptionalElse $\rightarrow$ λ

Cond $\rightarrow$ Expr gt Expr

Cond $\rightarrow$ Expr lt Expr

AssignStat $\rightarrow$ id assign Expr semicolon

Blue symbols are terminals.
Black symbols are nonterminals.
λ is just to show there is no symbol.

Examples of Context-Free Grammar Rules

Expr	$\rightarrow$	Expr plus Term
	|	Expr minus Term
	|	Term
Term	$\rightarrow$	Term mul Factor
	|	Term div Factor
	|	Factor
Factor	$\rightarrow$	number
	|	id
	|	lparen Expr rparen

"|" is used to group multiple rules for the same nonterminal.

Factor $\rightarrow$ number
Factor $\rightarrow$ id
Factor $\rightarrow$ lparen Expr rparen

Notation Adopted

Names Beginning With	**Examples**	**Represent Symbols in**
Upper case	A, B, C, Expr, Stmt	*N*
Lower case and punctuation	a, b, c, if, then, plus	*T*
X,*Y*	X_1, X_2	$N \cup T$
Other Greek letters	α, β, γ	$(N \cup T)^*$

The left hand side symbol of the first grammar rule is the start symbol unless stated otherwise.

Deriving Sentences (constructing sentences)

- A sequence of steps where nonterminals are replaced by the right-hand side of a rule is called ***a derivation***.
- Example of one sentence derived:

$\mathit{Expr} \Rightarrow \mathit{Expr}$ plus Term
$\Rightarrow \mathit{Term}$ plus Term
$\Rightarrow \mathit{Factor}$ plus Term
$\Rightarrow$ id plus Term
$\Rightarrow$ id plus Term mul Factor
$\Rightarrow$ id plus Factor mul Factor
$\Rightarrow$ id plus id mul Factor
$\Rightarrow$ id plus id mul id

One step derivation

Expr	→	Expr plus Term
	|	Expr minus Term
	|	Term
Term	→	Term mul Factor
	|	Term div Factor
	|	Factor
Factor	→	number
	|	id
	|	lparen Expr rparen

Deriving Sentences (constructing sentences)

- $\alpha \Rightarrow^* \beta$ means that β is derived in ***zero or more*** steps from α.

  Expr $\Rightarrow^*$ id plus *Term*, *Expr* $\Rightarrow^*$ id plus id mul id

- Since there may be multiple rules for a nonterminal, we may derive many different sentences from the same initial phrase.

  Example: *Expr* $\Rightarrow$ *Term*

  $\Rightarrow$ *Factor* mul *Factor*

  $\Rightarrow^*$ id mul id

- The set of terminal strings derivable from the start symbol, *S*, are the **set of all sentences** of the language, denoted ***L(G)***.

Deriving Sentences (constructing sentences)

- If there are multiple nonterminals in a phrase, there is a choice as to which nonterminal should be expanded next.

  Term $\Rightarrow$ ***Term*** mul ***Factor***

- In a ***leftmost*** derivation, we always expand the first nonterminal; In a ***rightmost*** one, the last nonterminal.

- Ultimately, the derivation order does not matter—we can derive any sentence in *L*(*G*) using any strategy.

- What is important, however, is which rule is applied at each nonterminal occurrence.

Term	$\rightarrow$	*Term* mul *Factor*
	|	*Term* div *Factor*
	|	*Factor*

Parse Trees (this is not the AST)

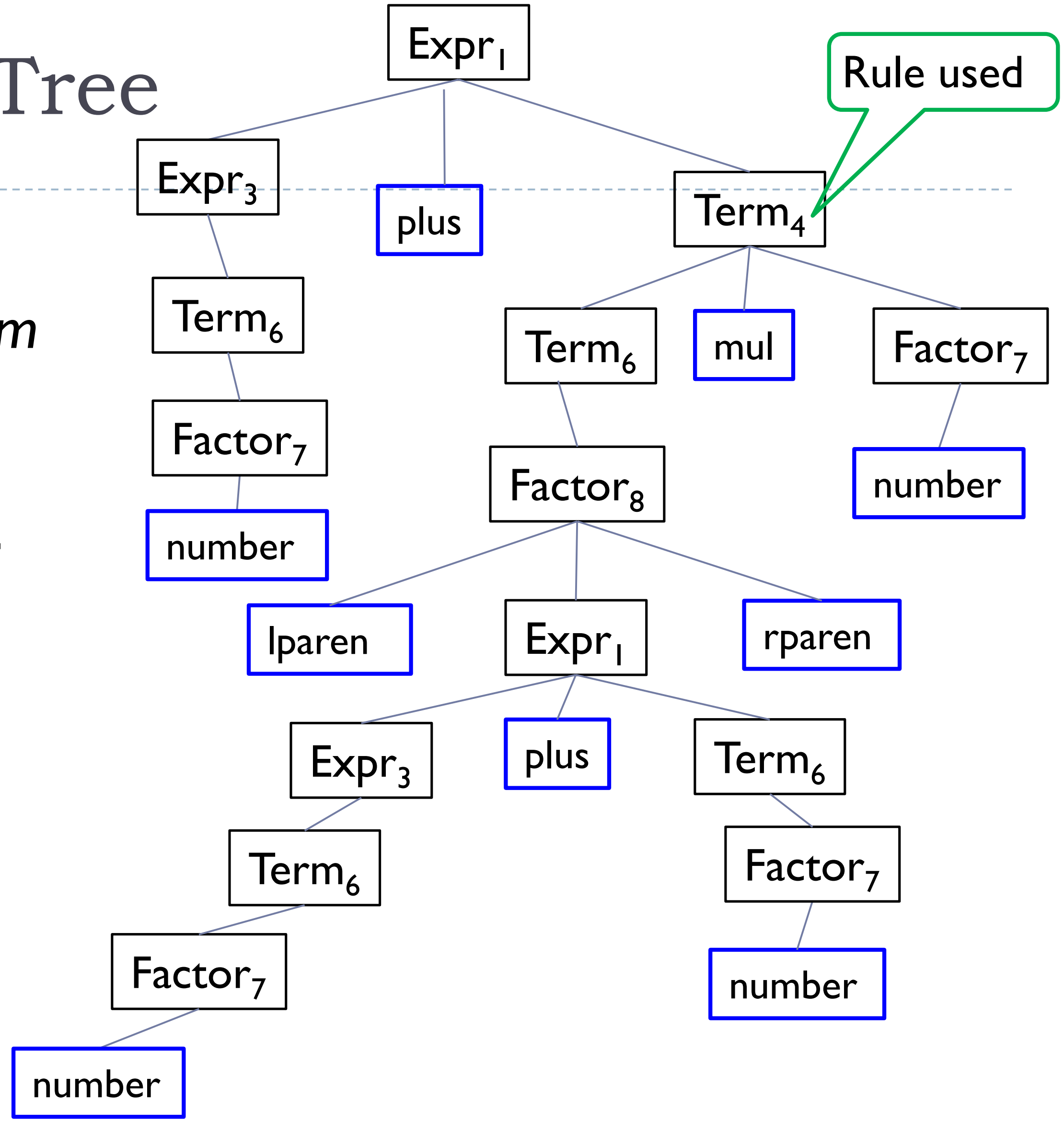
- A ***parse tree*** **represents a derivation**, and is used to show the structure of the sentence.

- Every node in a parse tree is labelled with a symbol:
  - The **root** node is labelled with the **start** symbol.
  - **Leaf** nodes are labelled with **terminal** symbols or λ.
  - **Inner nodes** are labelled with **nonterminal** symbols.

- A parse tree is generated with the following requirement:
  A node labelled A has children labelled $s_1 \ldots s_n$, if and only if there is a rule $A \rightarrow s_1 \ldots s_n$.

Example: Parse Tree

1 *Expr* → *Expr* plus *Term*
2 | *Expr* minus *Term*
3 | *Term*
4 *Term* → *Term* mul *Factor*
5 | *Term* div *Factor*
6 | *Factor*
7 *Factor* → number
8 | lparen *Expr* rparen



Sentence derived:

number plus lparen number plus number rparen mul number

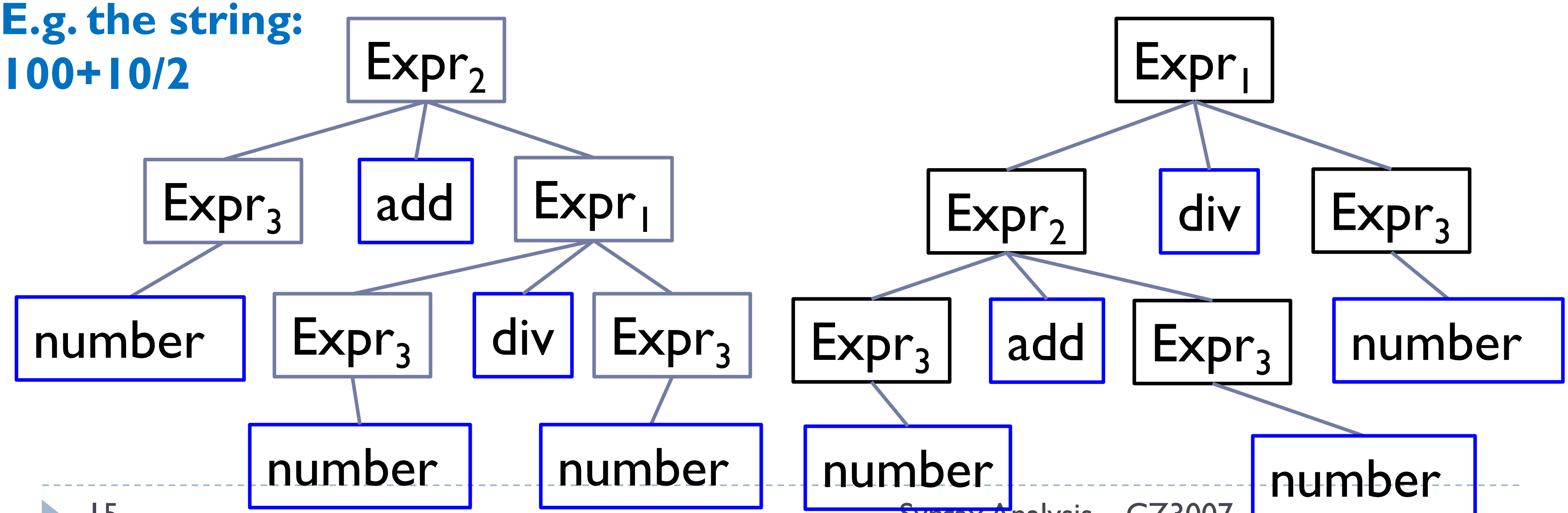
Ambiguous Grammar

- Consider the grammar *G* defined by the following 3 rules:

Expr	$\rightarrow$	Expr div Expr	// rule 1
	|	Expr add Expr	// rule 2
	|	number	// rule 3

Two parse trees for number add number div number:

E.g. the string: 100+10/2



Ambiguous Grammar

- **A grammar is ambiguous if it has multiple parse trees for one or more sentences. We can find more than one leftmost derivation or more than one rightmost derivation for such a sentence.**
- The meaning of such a sentence will be ambiguous.
- There is no algorithm that can check an arbitrary context-free grammar for ambiguity.

Test Yourself 3.1

Write a context-free grammar that defines a language of bit expressions involving bit operators AND, OR, XOR and COMPLEMENT. Tokens of the language are ONE, ZERO, AND, OR, XOR and COMPLEMENT, LPAREN, RPAREN. Examples of the bit expressions: 0, 0 OR ~1, 0 OR 1 AND 1, 1 AND (0 OR 0).

The Parsing Problem

Formal Statement
Given a context-free grammar $G = (T, N, S, P)$ and a sentence $w \in T^*$, decide whether or not $w \in L(G)$.

- ***Top-down parsing***: Generates a parse tree by starting at the **root** of the tree (the **start** symbol ***S***) and expanding the tree by applying rules in a depth-first manner.
- ***Bottom-up parsing***: Generates a parse tree by starting at the tree's **leaves** (and *w*) and working towards its root. A node is inserted into the tree only after its children have been inserted.

Top-Down Parsing

- This parsing technique is known by a few names:
  1. **Top-down**, because it begins with the grammar's start symbol and grows a parse tree from its root to its leaves.
  2. **Predictive**, because it predicts at each step which grammar rule is to be used.
  3. **LL(k)**, because it scans the input from Left to right, producing a Leftmost derivation, using k symbols of lookahead. We will consider only **LL(1)**.
  4. **Recursive descent**, because it can be implemented by a collection of mutually recursive procedures.

Recursive Descent Parsing

- In a recursive descent parser, for every nonterminal A there is a corresponding method parseA that can parse sentences derived from A.

The grammar:

$S \rightarrow A\ C\ \$$

$C \rightarrow c$

$\quad |\ \lambda$

$A \rightarrow a\ B\ C\ d$

$\quad |\ B\ Q$

$B \rightarrow b\ B$

$\quad |\ \lambda$

$Q \rightarrow q$

$\quad |\ \lambda$

The corresponding methods:

parseS {…}

parseC {…}

parseA {…}

parseB {…}

parseQ {…}

Example

The parser is implemented by a collection of mutually recursive procedures – recursive descent parser (slide 18).

```
parseS(ts)
{  // ts is the input token stream
   if (ts.peek() ∈ predict(p1))
          parseA(ts);  parseC(ts);
    else /* syntax error */        }

parseA(ts)
{  if (ts.peek() ∈ predict(p4))
          match(a);   parseB(ts);
          parseC(ts);   match(d);
   else if (ts.peek() ∈ predict(p5))
          parseB(ts);  parseQ(ts);
   else /* syntax error */        }

/* peek() examines the next input token
   without advancing the input  */
```

$S \to A\ C$		p_1
$C \to c$		p_2
$\quad \mid\ \lambda$		p_3
$A \to a\ B\ C\ d$		p_4
$\quad \mid\ B\ Q$		p_5
$B \to b\ B$		p_6
$\quad \mid\ \lambda$		p_7
$Q \to q$		p_8
$\quad \mid\ \lambda$		p_9

```
/* match(t) checks
ts.peek() == t  */
```

Recursive Descent Parsing

- The parsing of the whole program starts from the parse method for the start symbol.
- If there is more than one rule for A, parseA inspects the next input token and chooses a production rule among the rules for A to apply.
- The code for applying a production rule is obtained by processing the RHS of the rule, symbol by symbol:

  $A \rightarrow X_1 X_2 \ldots X_m$
- If the next symbol X_i is a terminal *t*, check whether the next input token is *t*.
- If it is a nonterminal B, call the parsing function parseB.
- The code for applying a production rule $A \rightarrow \lambda$ will do nothing and simply return.

Recursive Descent Parsing

- Recursive descent parsers use one token of lookahead to determine which rule to use.
- Lookahead has to be unambiguous—there should not be more than one rule (for the same nonterminal) whose RHS starts with the same token.

```
parseS(ts)
{  // ts is the input token stream
   if (ts.peek() ∈ predict(p1))
          parseA(ts);  parseC(ts);
    else /* syntax error */       }
parseA(ts)
{  if (ts.peek() ∈ predict(p4))
          match(a);   parseB(ts);
          parseC(ts);   match(d);
   else if (ts.peek() ∈ predict(p5))
          parseB(ts);  parseQ(ts);
   else /* syntax error */        }
```

Recursive Descent Parsing

- A grammar that fulfills this condition is an ***LL*(1) *grammar***.
- A language for which there exists an *LL*(1) grammar is an ***LL*(1) *language***.

Computing predict(p)

Consider a grammar rule p: $X \rightarrow X_1X_2\ldots X_m$, $m \geq 0$.

Each Xi is a terminal or non-terminal symbol

- The set of tokens that predict(p) returns includes

1) The set of first tokens in sentences derivable from $X_1X_2\ldots X_m$:
   - The set of first tokens in X_1
   - If X_1 may be empty, the set of first tokens in X_2, and so on.

2) If $X_1X_2\ldots X_m$ may be empty, the set of first tokens that may follow X

$S \rightarrow A\ C$	p_1
$C \rightarrow c$	p_2
$\mid\ \lambda$	p_3
$A \rightarrow$ **a B C d**	p_4
$\mid$ **B Q**	p_5
$B \rightarrow b\ B$	p_6
$\mid\ \lambda$	p_7
$Q \rightarrow q$	p_8
$\mid\ \lambda$	p_9

Computing predict(p)

Example 1: p: **A → B Q**

1) The set of first tokens in sentences derivable from B Q is {b, q}.
2) Because B Q may be empty, the set of first tokens that may follow A is {c, $}.

The set of tokens that predict(p) returns: {b, q, c, $}

S	→	A C	p_1
C	→	c	p_2
	|	λ	p_3
A	→	**a B C d**	p_4
	|	**B Q**	p_5
B	→	b B	p_6
	|	λ	p_7
Q	→	q	p_8
	|	λ	p_9

Example 2: p: **A → a B C d**

The set of tokens that predict(p) returns: {a}

The predict() results determine the rule to use

```
// predict(p4) = {a}
// predict(p5) = {b, q, c, $)
parseS(ts) { … }
parseA(ts)
{  if (ts.peek() ∈ predict(p4))
        match(a);  parseB(ts);
        parseC(ts);  match(d);
   else if (ts.peek() ∈ predict(p5))
        parseB(ts); parseQ(ts);
   else /* syntax error */      }
```

$S \rightarrow A\ C$	p_1
$C \rightarrow c$	p_2
$\quad \mid\ \lambda$	p_3
$A \rightarrow \mathbf{a\ B\ C\ d}$	p_4
$\quad \mid\ \mathbf{B\ Q}$	p_5
$B \rightarrow b\ B$	p_6
$\quad \mid\ \lambda$	p_7
$Q \rightarrow q$	p_8
$\quad \mid\ \lambda$	p_9

Computing predict(p)

- To compute the set of tokens that predict rule p, we need to know whether or not:
  - A nonterminal can derive empty
  - The RHS of a rule can derive empty
- Two boolean arrays are used:
  - symbolDerivesEmpty[X] for $X \in N$
  - ruleDerivesEmpty[p] for p $\in P$

$S \rightarrow A\ C$		p_1
$C \rightarrow c$		p_2
$\quad \mid\ \lambda$		p_3
$A \rightarrow$ **a B C d**		p_4
$\quad \mid$ **B Q**		p_5
$B \rightarrow b\ B$		p_6
$\quad \mid\ \lambda$		p_7
$Q \rightarrow q$		p_8
$\quad \mid\ \lambda$		p_9

Computing predict(p)

- For the grammar on the right
  - symbolDerivesEmpty[X] for $X \in N$:

S	C	A	B	Q
T	T	T	T	T

  - ruleDerivesEmpty[p] for $p \in P$:

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
T	F	T	F	T	F	T	F	T

$S \rightarrow$	A C	p_1
$C \rightarrow$	c	p_2
|	λ	p_3
$A \rightarrow$	**a B C d**	p_4
|	**B Q**	p_5
$B \rightarrow$	b B	p_6
|	λ	p_7
$Q \rightarrow$	q	p_8
|	λ	p_9

Computing predict(p)

predict(p: $X \rightarrow X_1X_2\ldots X_m$) // returns a set of tokens

{ $\quad ans$ = **first**($X_1X_2\ldots X_m$);

$\quad$ if ruleDerivesEmpty[p] then // when $X_1X_2\ldots X_m$ may be empty

$\quad\quad ans = ans \cup$ **follow**(X);

$\quad$ return ans;

}

- **first**($X_1X_2\ldots X_m$) returns the set of first tokens in sentences derivable from $X_1X_2\ldots X_m$. Formally:

  $$\text{first}(X_1X_2\ldots X_m) = \{t \in T \mid \exists w \in T^*, [X_1X_2\ldots X_m \Rightarrow^* tw]\}$$

Computing first($X_1X_2...X_m$)

first($X_1X_2...X_m$) // returns a set of tokens

{ for each nonterminal X in the language

visitedFirst[X] = false;

ans = **internalFirst**($X_1X_2...X_m$);

return *ans*;

}

Computing first($X_1X_2 \ldots X_m$)

The main ideas of **internalFirst**($X_1X_2 \ldots X_m$):

1. If $X_1X_2 \ldots X_m = \lambda$, there is no first token. Return empty set.

   $\boxed{\text{internalFirst}(\lambda) \text{ returns } \varnothing}$

2. If X_1 is a terminal symbol, the first token is this symbol. Return $\{X_1\}$.

   $\boxed{\text{internalFirst(b B) returns b}}$

Computing first($X_1X_2 \ldots X_m$)

3. If X_l is a nonterminal:
   1) If visitedFirst[X_l] is true, return $\varnothing$. Else
   2)
      i. Set visitedFirst[X_l] to true.
      ii. Look at every rule for X_l and find the first tokens of each rule and add to the result.
      iii. If symbolDerivesEmpty[X_l], find the first tokens for $X_2 \ldots X_m$ and add to the result.

Why every rule?

If we change rule p_5 to "A $\rightarrow$ B Q a B" in slide 26, internalFirst(B Q a B) returns {b, q, a}.

Computing follow(X)

- **follow**(X) returns a set of tokens that can appear right behind the nonterminal X in a phrase derived from the start symbol *S*. Formally:

  $\text{follow}(X) = \{t \in T \mid \exists \alpha,\beta \in (N \cup T)^*, [S \Rightarrow^* \alpha X t \beta\ ]\}$

follow(*X*) // returns a set of tokens that may follow *X*

{   for each nonterminal *Y* in the language

         visitedFollow[*Y*] = false;

     ans = **internalFollow**(*X*);

     return *ans*;

}

Main Ideas of internalFollow(X)

1) If visitedFollow[X] is true, return $\varnothing$. Else

2) (i) Set visitedFollow[X] to true.

   (ii) Find all occurrences of X in all RHS,

   E.g. All occurences of B shown in blue

   (iii) For each such occurrence, find the **first tokens** of the sequence after X.
   If the sequence after X may derive empty, call internalFollow(LHS) to find what tokens follow the LHS nonterminal.

   (iv) If X is the start symbol, add $ to the result.

$S \rightarrow A\ C$

$C \rightarrow c$

$\quad\mid\ \lambda$

$A \rightarrow a\ \mathbf{B}\ C\ d$

$\quad\mid\ \mathbf{B}\ Q$

$B \rightarrow b\ \mathbf{B}$

$\quad\mid\ \lambda$

$Q \rightarrow q$

$\quad\mid\ \lambda$

Example: internalFollow(B)

1. Occurrence of B in ‘A $\rightarrow$ a B C d’:
   String after B = ‘C d’
   first(C d) returns {c, d}
   Result so far = {c, d}
2. Occurrence of B in ‘A $\rightarrow$ B Q’:
   String after B = ‘Q’
   first(Q) returns {q}
   Q may be empty
   internalFollow(A) returns {c, $}
   Result so far = {c, d, q, $}
3. Occurrence of B in ‘B $\rightarrow$ b B’:
   string after B is empty
   internalFollow(LHS) returns $\emptyset$. **Final result = {c, d, q, $}**

$$
\begin{aligned}
S &\rightarrow A\ C \\
C &\rightarrow c \\
&\quad |\ \lambda \\
A &\rightarrow a\ \mathbf{B}\ C\ d \\
&\quad |\ \mathbf{B}\ Q \\
B &\rightarrow b\ \mathbf{B} \\
&\quad |\ \lambda \\
Q &\rightarrow q \\
&\quad |\ \lambda
\end{aligned}
$$

Test Yourself 3.2

Q1.

1. $S \rightarrow A \;\; B \;\; c$
2. $A \rightarrow a$
3. $\quad | \; \lambda$
4. $B \rightarrow b$
5. $\quad | \; \lambda$

predict($S \rightarrow A \;\; B \;\; c$) = ?

Q2

1. $S \rightarrow A \;\; B \;\; c$
2. $A \rightarrow a$
3. $\quad | \; \lambda$
4. $B \rightarrow b$
5. $\quad | \; \lambda$

predict($A \rightarrow \lambda$) = ?

predict($B \rightarrow \lambda$) = ?

Obtaining LL(1) Grammars

- LL(1) requires a unique combination of a nonterminal and a lookahead symbol to decide which rule to use.
- Two common categories of grammar rules make a grammar not LL(1)—*common prefixes* and *left recursion*.

Common Prefixes

- If the RHSs of two rules for the same nonterminal start with the same lookahead symbol, the grammar is not LL(1).

Example
Expr → number plus *Expr* | number

Obtaining LL(1) Grammars

Example
Expr $\rightarrow$ number plus *Expr* | *Factor*
Factor $\rightarrow$ number

- One way to eliminate common prefixes is by removing the common left factor and introducing new nonterminals (left factoring a grammar):

Example
Expr $\rightarrow$ number *Expr'*
Expr' $\rightarrow$ plus Expr | λ

This grammar accepts the same language as the one on the previous slide, and this language is LL(1).

Obtaining LL(1) Grammars

Left Recursion

- If the RHS of a rule starts with the LHS nonterminal, the grammar is not LL(1):

Example
StmtList $\rightarrow$ *StmtList* semicolon *Stmt*
$\quad\quad\quad\quad$ | *Stmt*
…

The predict() results from the two rules will be the same.

E.g. A $\rightarrow$ A c
$\quad\quad$ | c $\quad\quad\quad\quad$ // predict() for both rules give c

Obtaining LL(1) Grammars

1. Change left recursion to right recursion:

$$\begin{aligned} StmtList &\rightarrow Stmt \text{ semicolon } StmtList \\ &\mid Stmt \end{aligned}$$

2. Remove the common prefix:

$$\begin{aligned} StmtList &\rightarrow Stmt\ \ StmtListTail \\ StmtListTail &\rightarrow \text{semicolon } StmtList \\ &\mid \lambda \end{aligned}$$

3. May want to remove mutual recursion (optional):

$$\begin{aligned} StmtList &\rightarrow Stmt\ \ StmtListTail \\ StmtListTail &\rightarrow \text{semicolon } Stmt\ \ StmtListTail \\ &\mid \lambda \end{aligned}$$

Syntactic Error Recovery

- A compiler should produce a useful <u>set</u> of diagnostic messages when presented with a faulty program.
- Semantic analysis and code generation will be disabled.
- After an error is detected, it is desirable to recover from it and continue the syntax analysis.
- In a simple form of error recovery, the parser skips input tokens until it finds a delimiter (e.g., a semicolon) to end the parsing of the current nonterminal.
- The method for parsing a nonterminal is augmented with an extra parameter that is a set of delimiters.

Example

first(d e)

```
parseA(ts, termset)
{  // ts is the input token stream
   if (ts.peek() ∈ {a})
         match(a);   parseB(ts, {d} ∪ termset);
         match(d);   match(e);
   else if (ts.peek() ∈ {b})
         parseB(ts, {q,e} ∪ termset);
         parseQ(ts, {e} ∪ termset);  match(e);
   else
         error("expected an a or b");
         skip input till a symbol in termset is found
}
```

first(Q e)

$A \rightarrow a\ B\ d\ e$

$\quad |\ B\ Q\ e$

$B \rightarrow b$

$Q \rightarrow q$

$\quad |\ \lambda$

End-of-file symbol is in the *termset* of every parsing method.

The Parsing Problem

Formal Statement
Given a context-free grammar $G = (T, N, S, P)$ and a sentence $w \in T^*$, decide whether or not $w \in L(G)$.

- ***Top-down parsing***: Generates a parse tree by starting at the **root** of the tree (the **start** symbol ***S***) and expanding the tree by applying rules in a depth-first manner.

- ***Bottom-up parsing***: Generates a parse tree by starting at the tree's **leaves** (and *w*) and working towards its root. A node is inserted into the tree only after its children have been inserted.

Bottom-Up Parsing

- Bottom-up parsing starts by generating the leaves of a parse tree. A node is inserted into the tree only after its children have been inserted.
- Recursive-descent (Top-down) parsers are quite versatile and appropriate for a hand-written parser.
- Bottom-up parsers are commonly used in the syntax analysis phase of a compiler because of its power, efficiency and ease of construction.
- Grammar features like common prefixes and left recursion need to be addressed before top-down parsing can be used. But they can be accommodated without issue in bottom-up parsing.

Bottom-Up Parsing

- This parsing technique is known by a few names:
  1. **Bottom-up**, because it works its way from the terminal symbols to the grammar's start symbol.
  2. **Shift-reduce**, because the two prevalent actions taken by the parser are to *shift* symbols onto the parse stack and to *reduce* a string of such symbols at the top-of-stack to one of the grammar's nonterminals.
  3. **LR(k)**, because it scans the input from left to right, producing a rightmost derivation in reverse, using k symbols of lookahead.

Rightmost Derivation in Reverse

Rule	Derivation/construction
1	Start $\Rightarrow$ S $
2	$\Rightarrow$ A C $
3	$\Rightarrow$ A c $
5	$\Rightarrow$ a B C d c $
4	$\Rightarrow$ a B d c $
7	$\Rightarrow$ a b B d c $
7	$\Rightarrow$ a b b B d c $
8	$\Rightarrow$ a b b d c $

The bottom up parser will read the sentence and use the rules in reverse order.

1. Start $\rightarrow$ S $
2. S $\rightarrow$ A C
3. C $\rightarrow$ c
4. | λ
5. A $\rightarrow$ a B C d
6. | B Q
7. B $\rightarrow$ b B
8. | λ
9. Q $\rightarrow$ q
10. | λ

LR Parsing Engine

- The parsing engine is driven by a table.
- It can also be described as a finite automaton, an algorithm that can recognise (accept) all sentences of a language and reject those which do not belong to them.
- The table is indexed using the parser's *current state* and the *next input symbol*.
- At each step, the engine looks up the table based on the current state and the next input symbol for an action.
- The table entry indicates the action to perform (either a shift or a reduce, till the final action which is accept).

back

```
call Stack.PUSH( StartState )
accepted ← false
while not accepted do
    action ← Table[Stack.TOS( )][InputStream.PEEK( )]          (1)
    if action = shift s
    then
        call Stack.PUSH( s )                                     (2)
        if s ∈ AcceptStates                                      (3)
        then  accepted ← true
        else  call InputStream.ADVANCE( )
    else
        if action = reduce A → γ
        then                        // apply rule A → γ
            call Stack.POP( |γ| )                                (4)
            call InputStream.PREPEND( A )                        (5)
        else
            call ERROR( )                                        (6)
```

Figure 6.3: Driver for a bottom-up parser.

1. Start $\rightarrow$ S \$
2. S $\rightarrow$ A C
3. C $\rightarrow$ c
4. | λ
5. A $\rightarrow$ a B C d
6. | B Q
7. B $\rightarrow$ b B
8. | λ
9. Q $\rightarrow$ q
10. | λ

shift state 3

Reduce by rule 8

State	a	b	c	d	q	\$	Start	S	A	B	C	Q
0	$\boxed{3}$	$\boxed{2}$	8		8	8	accept	$\boxed{4}$	$\boxed{1}$	$\boxed{5}$		
1			$\boxed{11}$			4					$\boxed{14}$	
2		$\boxed{2}$	8	8	8	8				$\boxed{13}$		
3		$\boxed{2}$	8	8						$\boxed{9}$		
4						$\boxed{8}$						
5			10		$\boxed{7}$	10						$\boxed{6}$
6			6			6						
7			9			9						
8						1						
9			$\boxed{11}$	4							$\boxed{10}$	
10				$\boxed{12}$								
11				3		3						
12			5			5						
13			7	7	7	7						
14						2						

Figure 6.5: Parse table for the grammar

shift state 3

Reduce by rule 8

input: a b b d c $

0

input: b b d c $

0	3
	a

input: b d c $

0	3	2
	a	b

input: d c $

0	3	2	2
	a	b	b

State	a	b	c	d	q	$	Start	S	A	B	C	Q
0	$\boxed{3}$	$\boxed{2}$	8		8	8	accept	$\boxed{4}$	$\boxed{1}$	$\boxed{5}$		
1			$\boxed{11}$			4					$\boxed{14}$	
2		$\boxed{2}$	8	8	8	8				$\boxed{13}$		
3		$\boxed{2}$	8	8						$\boxed{9}$		
4						$\boxed{8}$						
5			10		$\boxed{7}$	10						$\boxed{6}$
6			6			6						
7			9			9						
8						1						
9			$\boxed{11}$	4							$\boxed{10}$	
10				$\boxed{12}$								
11				3		3						
12			5			5						
13			7	7	7	7						
14						2						

Figure 6.5: Parse table for the grammar

shift state 3

Reduce by rule 8

8. B → λ

input: B d c $

0	3	2	2
	a	b	b

input: d c $

0	3	2	2	13
	a	b	b	B

7. B → b B

input: B d c $

0	3	2
	a	b

input: d c $

0	3	2	13
	a	b	B

State	a	b	c	d	q	$	Start	S	A	B	C	Q
0	$\boxed{3}$	$\boxed{2}$	8		8	8	accept	$\boxed{4}$	$\boxed{1}$	$\boxed{5}$		
1			$\boxed{11}$			4					$\boxed{14}$	
2		$\boxed{2}$	8	8	8	8				$\boxed{13}$		
3		$\boxed{2}$	8	8						$\boxed{9}$		
4						$\boxed{8}$						
5			10		$\boxed{7}$	10						$\boxed{6}$
6			6			6						
7			9			9						
8						1						
9			$\boxed{11}$	4							$\boxed{10}$	
10				$\boxed{12}$								
11				3		3						
12			5			5						
13			7	7	7	7						
14						2						

Figure 6.5: Parse table for the grammar

shift state 3

Reduce by rule 8

7. B → b B

input: B d c $

0	3
	a

input: d c $

0	3	9
	a	B

4. C → λ

input: C d c $

0	3	9
	a	B

input: d c $

0	3	9	10
	a	B	C

State	a	b	c	d	q	$	Start	S	A	B	C	Q
0	$\boxed{3}$	$\boxed{2}$	8		8	8	accept	$\boxed{4}$	$\boxed{1}$	$\boxed{5}$		
1			$\boxed{11}$			4					$\boxed{14}$	
2		$\boxed{2}$	8	8	8	8				$\boxed{13}$		
3		$\boxed{2}$	8	8						$\boxed{9}$		
4						$\boxed{8}$						
5			10		$\boxed{7}$	10						$\boxed{6}$
6			6			6						
7			9			9						
8						1						
9			$\boxed{11}$	4							$\boxed{10}$	
10				$\boxed{12}$								
11				3		3						
12			5			5						
13			7	7	7	7						
14						2						

Figure 6.5: Parse table for the grammar

shift state 3

Reduce by rule 8

input: c $

0	3	9	10	12
	a	B	C	d

5. A → a B C d

input: A c $

0

input: c $

0	1
	A

input: $

0	1	11
	A	c

State	a	b	c	d	q	$	Start	S	A	B	C	Q
0	$\boxed{3}$	$\boxed{2}$	8		8	8	accept	$\boxed{4}$	$\boxed{1}$	$\boxed{5}$		
1			$\boxed{11}$			4					$\boxed{14}$	
2		$\boxed{2}$	8	8	8	8				$\boxed{13}$		
3		$\boxed{2}$	8	8						$\boxed{9}$		
4						$\boxed{8}$						
5			10		$\boxed{7}$	10						$\boxed{6}$
6			6			6						
7			9			9						
8						1						
9			$\boxed{11}$	4							$\boxed{10}$	
10				$\boxed{12}$								
11				3		3						
12			5			5						
13			7	7	7	7						
14						2						

Figure 6.5: Parse table for the grammar

shift state 3

Reduce by rule 8

3. C → c

input: C $

0	1 A

input: $

0	1 A	14 C

2. S → A C

input: S $

0

input: $

0	4 S

State	a	b	c	d	q	$	Start	S	A	B	C	Q
0	$\boxed{3}$	$\boxed{2}$	8		8	8	accept	$\boxed{4}$	$\boxed{1}$	$\boxed{5}$		
1			$\boxed{11}$			4					$\boxed{14}$	
2		$\boxed{2}$	8	8	8	8				$\boxed{13}$		
3		$\boxed{2}$	8	8						$\boxed{9}$		
4						$\boxed{8}$						
5			10		$\boxed{7}$	10						$\boxed{6}$
6			6			6						
7			9			9						
8						1						
9			$\boxed{11}$	4							$\boxed{10}$	
10				$\boxed{12}$								
11				3		3						
12			5			5						
13			7	7	7	7						
14						2						

Figure 6.5: Parse table for the grammar

shift state 3

Reduce by rule 8

input: $

0	4	8
	S	$

1. Start → S $

input: Start $

0

Accept

a b b d c $

This bottom-up parsing applies the rules in reverse order to the one in slide 46

State	a	b	c	d	q	$	Start	S	A	B	C	Q
0	$\boxed{3}$	$\boxed{2}$	8		8	8	accept	$\boxed{4}$	$\boxed{1}$	$\boxed{5}$		
1			$\boxed{11}$			4					$\boxed{14}$	
2		$\boxed{2}$	8	8	8	8				$\boxed{13}$		
3		$\boxed{2}$	8	8						$\boxed{9}$		
4						$\boxed{8}$						
5			10		$\boxed{7}$	10						$\boxed{6}$
6			6			6						
7			9			9						
8						1						
9			$\boxed{11}$	4							$\boxed{10}$	
10				$\boxed{12}$								
11				3		3						
12			5			5						
13			7	7	7	7						
14						2						

Figure 6.5: Parse table for the grammar

Classes of Bottom-up Parsers

- All bottom-up parsers uses a single token of lookahead during parsing.
- They are finite automata with a stack.
- There are several methods for constructing parse tables. The number indicates the no. of lookahead tokens when building the parse table.
  - LR(0): Simplest, fails for many practical grammars.
  - LR(1): Quite general, can handle almost all practically interesting grammars.
  - LALR(1): Faster, slightly weaker variant of LR(1), used by most bottom-up parser generators.

LR(0) Automata and Table Construction

- The table construction process analyses the grammar.
- Each state corresponds to a row of the parser table.
- Each symbol in the terminal and nonterminal sets corresponds to a column of the table.

State	a	b	c	d	q	$	Start	S	A	B	C
0	$\boxed{3}$	$\boxed{2}$	8		8	8	accept	$\boxed{4}$	$\boxed{1}$	$\boxed{5}$	
1			$\boxed{11}$			4					$\boxed{14}$
2		$\boxed{2}$	8	8	8	8				$\boxed{13}$	
3		$\boxed{2}$	8	8						$\boxed{9}$	
4						$\boxed{8}$					
5			10		$\boxed{7}$	10					
6			6			6					
7			9			9					
8						1					
9			$\boxed{11}$	4							$\boxed{10}$
10				$\boxed{12}$							
11				3		3					
12			5			5					
13			7	7	7	7					
14						2					

Figure 6.5: Parse table example.

LR(0) Automata and Table Construction

- During parsing, we keep track of where we are in the grammar.
- To do this, we use a marker, •, in a grammar rule to show the current progress of the parser in recognising the RHS of the rule.
- Symbols before the • have already been seen. The first symbol after • is what we expect next. For example,

  E $\rightarrow$ • plus E E

  E $\rightarrow$ plus • E E

  E $\rightarrow$ plus E • E

LR(0) Automata and Table Construction

- LR(0) table has a number of states.
- Each state consists of a number of LR(0) items.
- **LR(0) items**: An LR(0) item is **a grammar rule with a marker •**.
- For an item $A \to \alpha \bullet \beta$, the item is called ***initial*** if $\alpha \to \lambda$, and ***final*** if $\beta \to \lambda$. A final item for the start symbol is called accepting.

E. g.

$Start \to \bullet\ S$ // an initial item

$S \to A\ C\ \bullet$ // a final item

$B \to \bullet\ \lambda$ // an initial item and a final item

LR(0) Automata and Table Construction

- Each LR(0) state is **a set of LR(0) items**, which is ***closed*** in the sense that if the state contains an item with a nonterminal A immediately after the marker, we add the initial items for A, into the state. This is called taking ***the closure of the item***.

E.g., For an item $S \rightarrow \bullet A$ in a state, we add into the state
$A \rightarrow \bullet a\ B\ d$, $A \rightarrow \bullet B\ Q$,
$B \rightarrow \bullet b$

i.e., closure = {$S \rightarrow \bullet A$, $A \rightarrow \bullet a\ B\ d$, $A \rightarrow \bullet B\ Q$, $B \rightarrow \bullet b$}

$S \rightarrow A$
$A \rightarrow a\ B\ d$
$\quad\ \mid B\ Q$
$B \rightarrow b$
$Q \rightarrow q$

LR(0) Automata and Table Construction

- We describe the parsing process as a finite automaton.
- The start state is the closure of the initial items of the start symbol:

  For the grammar on the right, there is only one initial item for the Start symbol:

  Start → • E $

Start → E $
E → plus E E
| num

Taking the closure, we get the set of items in state 0:

0

Start → • E $
E → • plus E E
E → • num

In slides 50, 53, 54 55, state 0 is on TOS—when in state 0, at diff. times, there are a few possible in-coming symbols.

LR(0) Automata and Table Construction

- For each symbol γ that appears to the right of the marker of an item (or items), we can **shift** over it and transition to a new state:
  - Replace all items where γ appears to the right of the marker by items where the marker follows γ.
  - Throw away all other items.
  - Take the closure of these items.

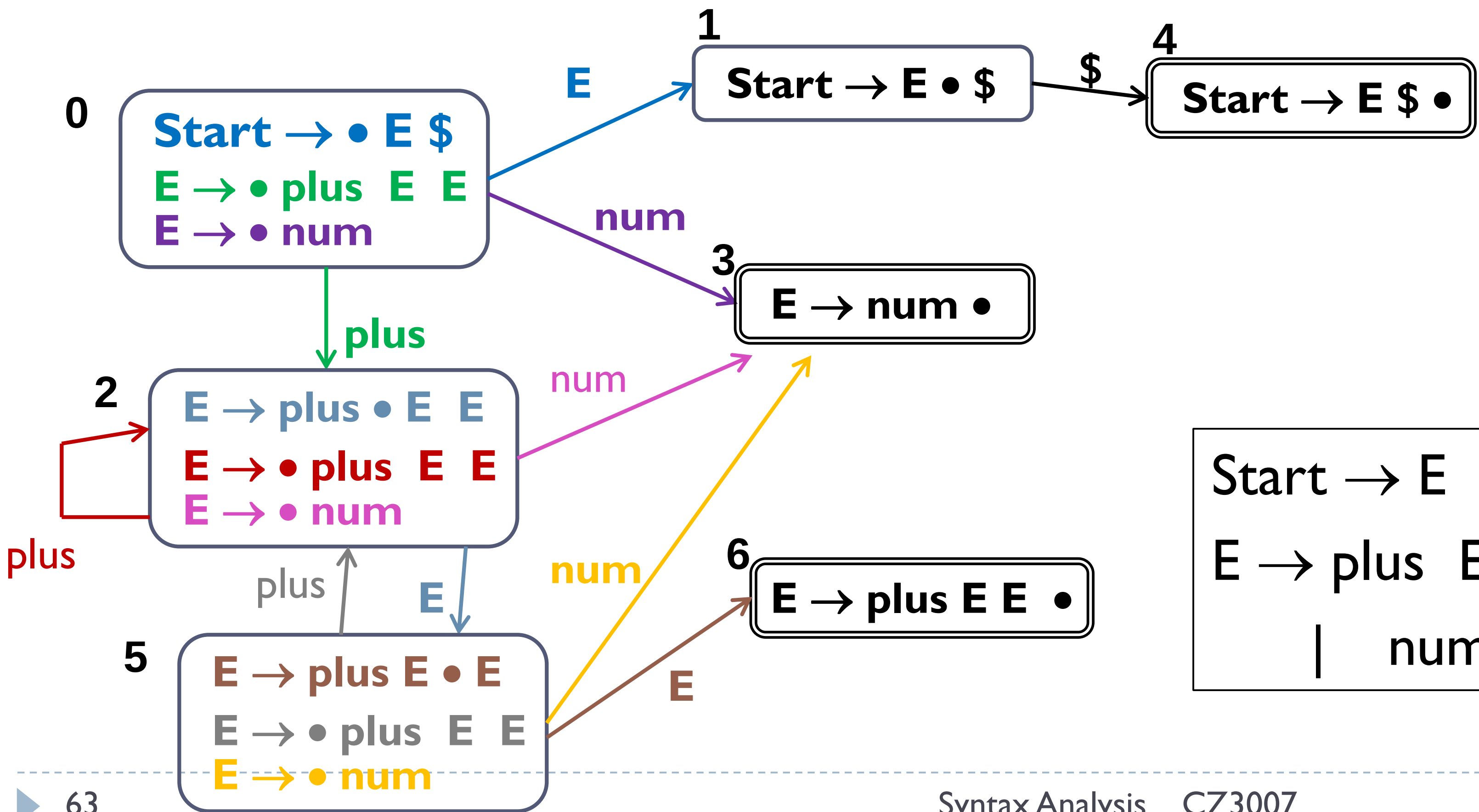
- Transitions are labelled by γ which is either a terminal or a nonterminal.

- If there is a final item of the form $A \rightarrow \alpha \bullet$, we can **reduce**.

- An accepting state is one that contains a final item for the start symbol.

LR(0) Automata and Table Construction

- Continuing from State 0 of our finite automata in slide 61, each of the three items will lead to a new state. Each new state leads to more new states.



Start → E $
E → plus E E
| num

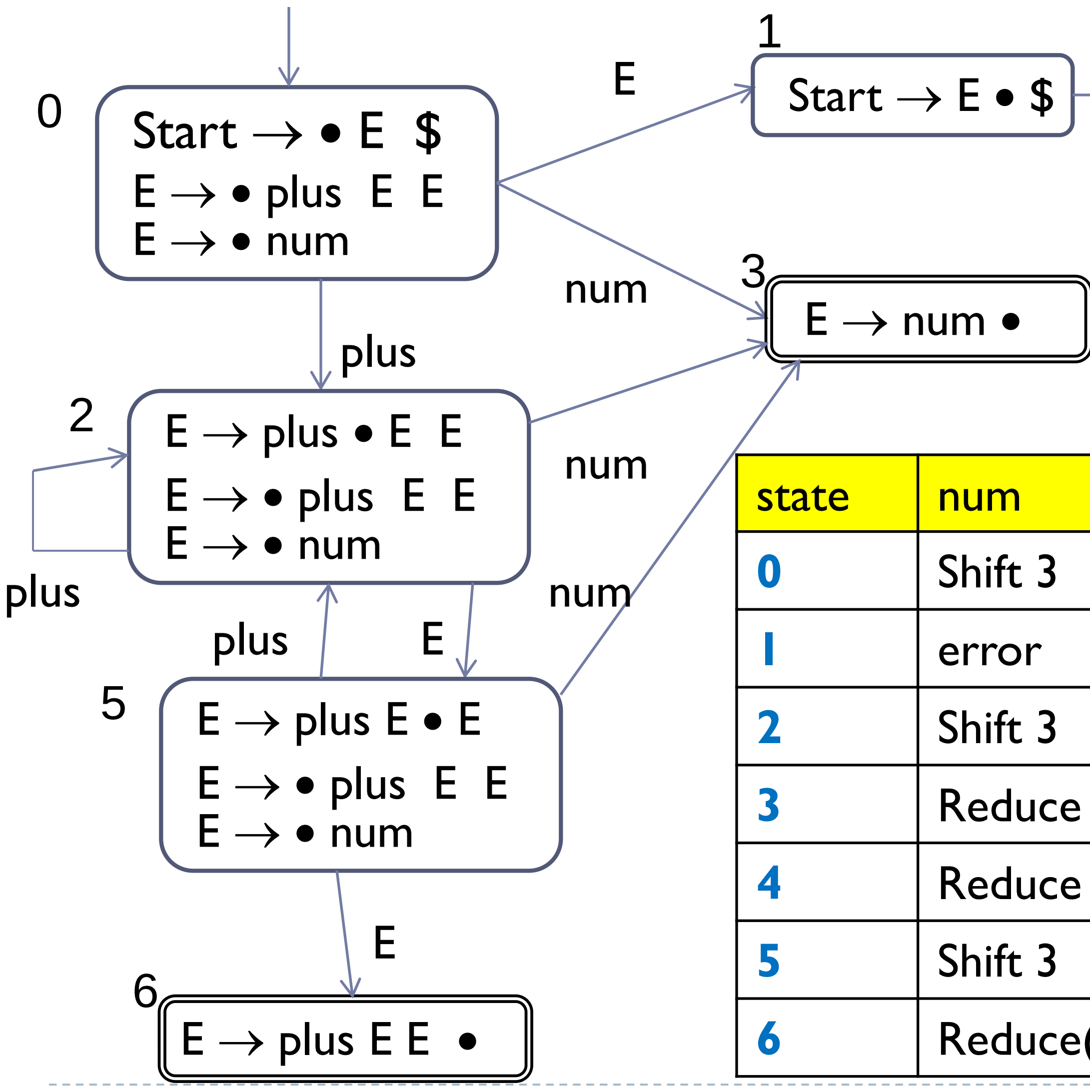
LR(0) Automata and Table Construction

- The LR(0) automaton needs to be used together with a stack of states; This kind of automaton is known as a ***pushdown automaton.***
- The state of the parser is the state on top of the parser stack.
- Actions taken during bottom-up parsing are of four types:
  - **shift**(i): Consume the next input symbol, push state i onto state stack
  - **reduce**($A \to \gamma$): Reduce by rule $A \to \gamma$, i.e., pop $|\gamma|$ states from the stack and consider A as next input symbol
  - **accept**: Report that input was parsed successfully
  - **error**: Report a parse error

LR(0) Automata and Table Construction

- An LR(0) parse table is a compact representation of an LR(0) automaton.
- Rows are indexed by states, columns by symbols; cell in row r , column c contain a single parsing action to take when encountering input symbol c in state r.
- Constructing a parse table from an LR(0) automaton is easy:
  - For every transition from a state s to a state s' labelled with a symbol x, enter **shift**(s') into the cell in row s, column x.
  - If state s contains a final item $A \rightarrow \beta \bullet$, enter **reduce**($A \rightarrow \beta$) into all cells of row s.
  - For cell (0, StartSymbol), enter **accept.**
  - Enter **error** into any remaining empty cells.

Example



Parser Table

state	num	plus	$	Start	E
0	Shift 3	Shift 2	error	accept	Shift 1
1	error	error	Shift 4	error	error
2	Shift 3	Shift 2	error	error	Shift 5
3	Reduce (E → num)				
4	Reduce (Start → E $)				
5	Shift 3	Shift 2	error	error	Shift 6
6	Reduce(E → plus E E)				

Test Yourself 3.3

Construct the LR(0) parse table for the following language grammar:

$$
\begin{array}{l}
\textbf{Start} \rightarrow \textbf{E}\ \ \$ \\
\textbf{E} \rightarrow \textbf{E}\ \ \textbf{E}\ \ \textbf{plus} \\
\quad\quad |\ \ \textbf{num}
\end{array}
$$

Conflicts

- Sometimes when trying to construct an LR(0) parse table we end up with two different actions in the same cell; this is known as a conflict.

- There are two kinds of conflicts:
  - **Shift-reduce conflict**: The same cell contains both a **shift()** action and a **reduce()** action.
  - **Reduce-reduce conflict**: The same cell contains two different **reduce()** actions.

- **Question: Can there be a shift-shift conflict?**

Shift-Reduce Conflict

- **Shift-reduce conflict**: If a state contains both a non-final item $A \to \beta \bullet \gamma$ (a **shift()** action) and a final item $A \to \beta \bullet$ (a **reduce()** action)

  E.g. A state has these two items:

  IfStatement $\to$ IF Cond THEN StatList $\bullet$ ELSE StatList

  IfStatement $\to$ IF Cond THEN StatList $\bullet$

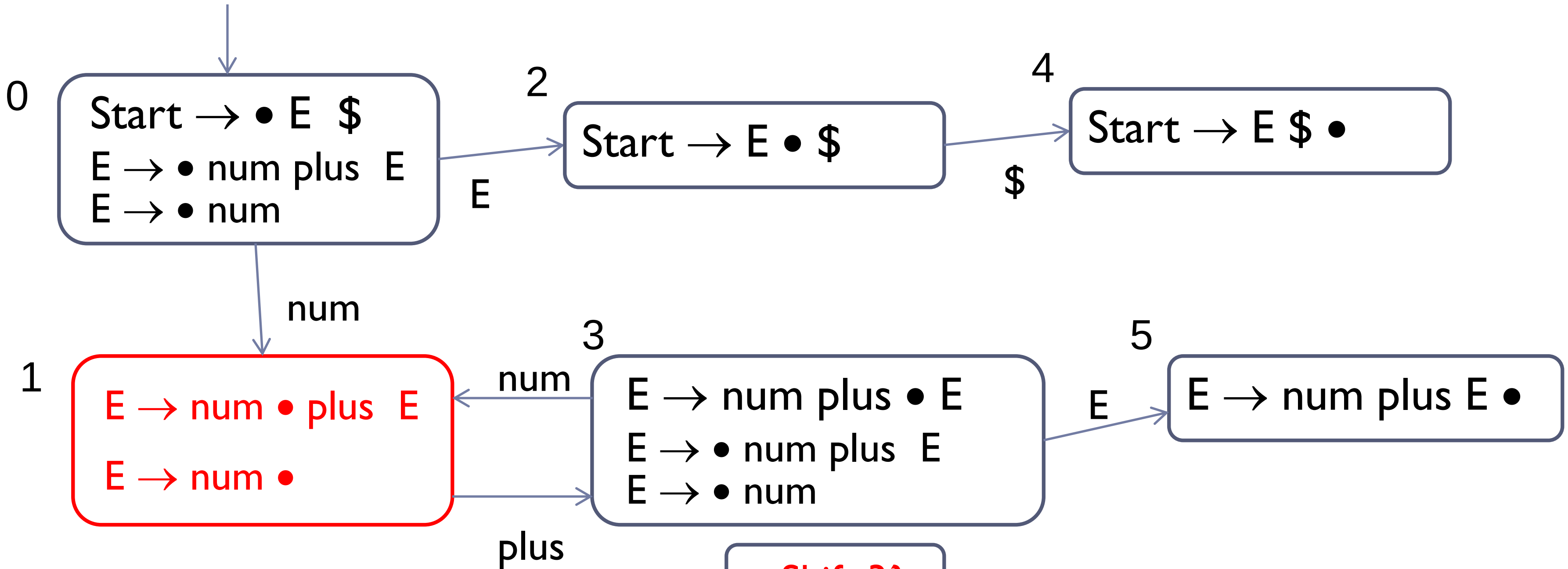
- Example:

  Start $\to$ E \$

  E $\to$ num plus E

  | num

Example: Shift-Reduce Conflict



Shift 3?

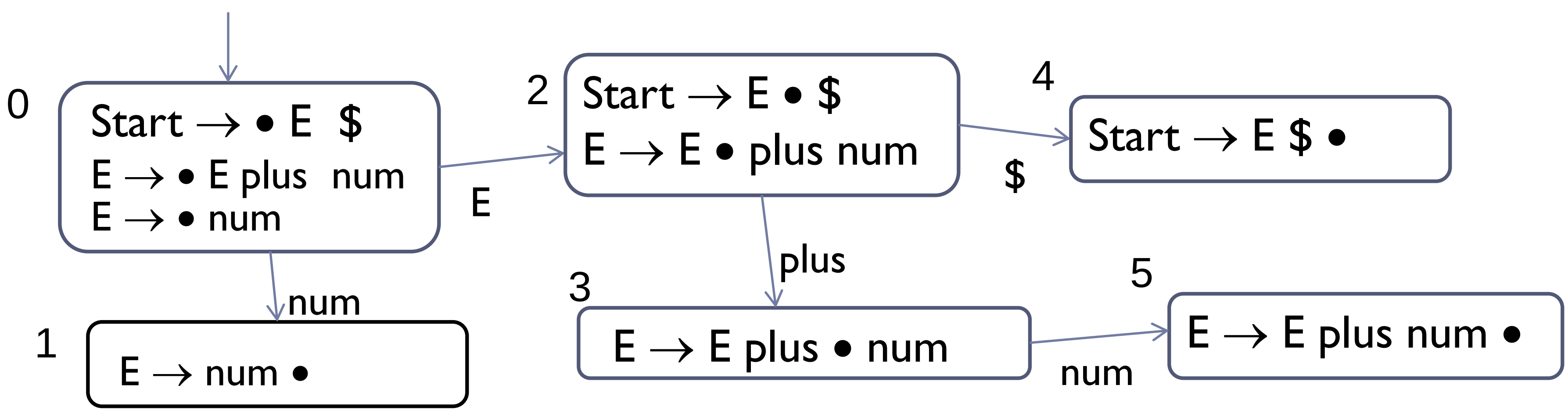
state	num	plus	$	Start	E
0	Shift 1	error	error	accept	Shift 2
1	Reduce (E → num)				
2	…				

Shift-Reduce Conflict

- A **shift-reduce conflict** may be eliminated by rewriting the grammar. For example,

Rewrite to

Start → E $	Start → E $
E → num plus E	E → E plus num
| num	| num



An Ambiguous Grammar

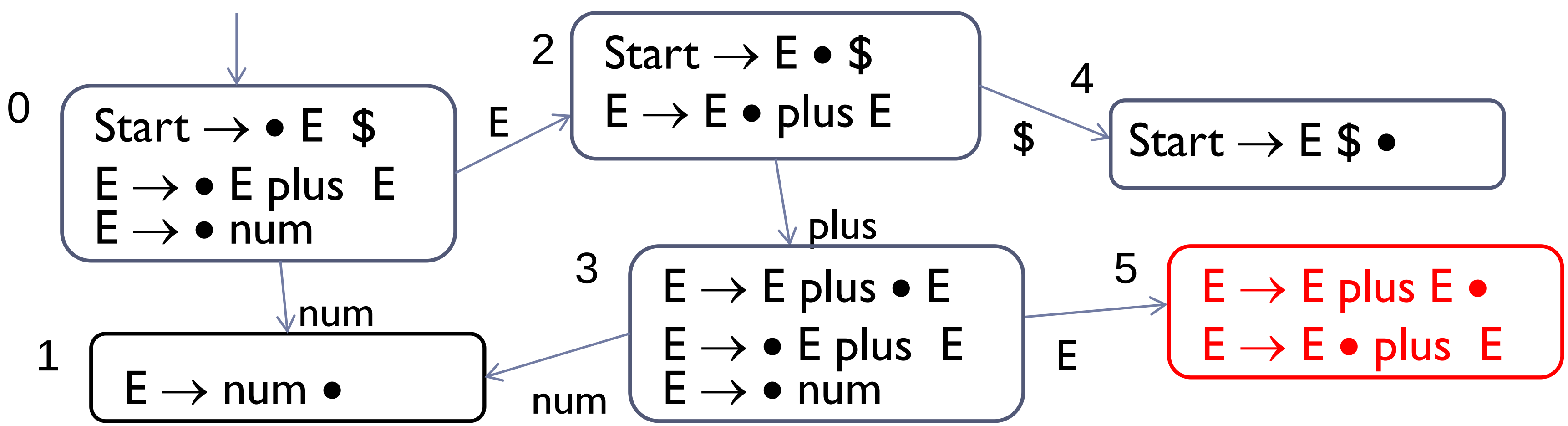
- Ambiguous grammars always lead to conflicts. For example,

  Start → E $

  E → E plus E

  | num

LR(0) automaton:

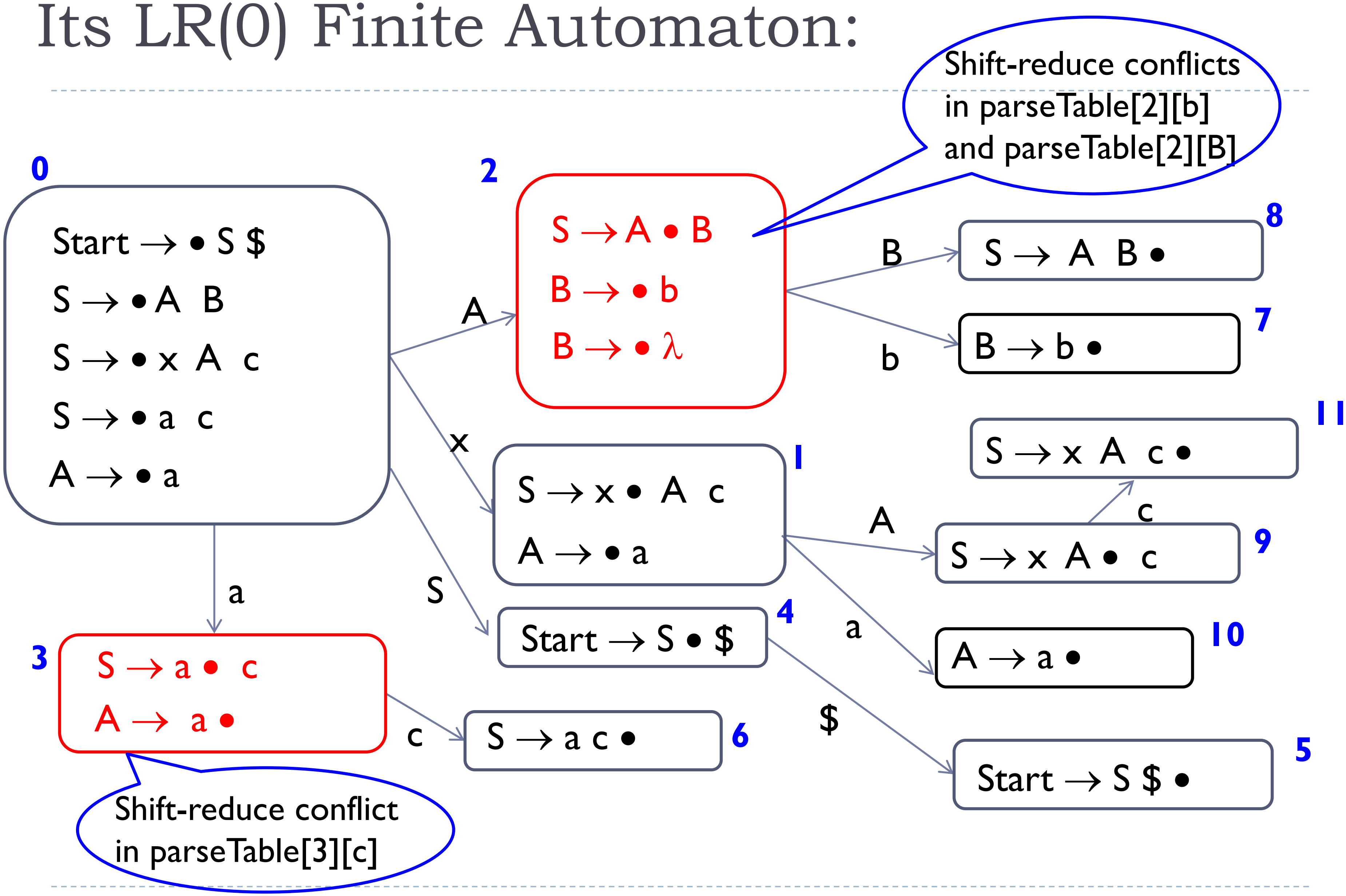


Rewriting the grammar will solve the problem sometimes.
An alternative is a more powerful parser.

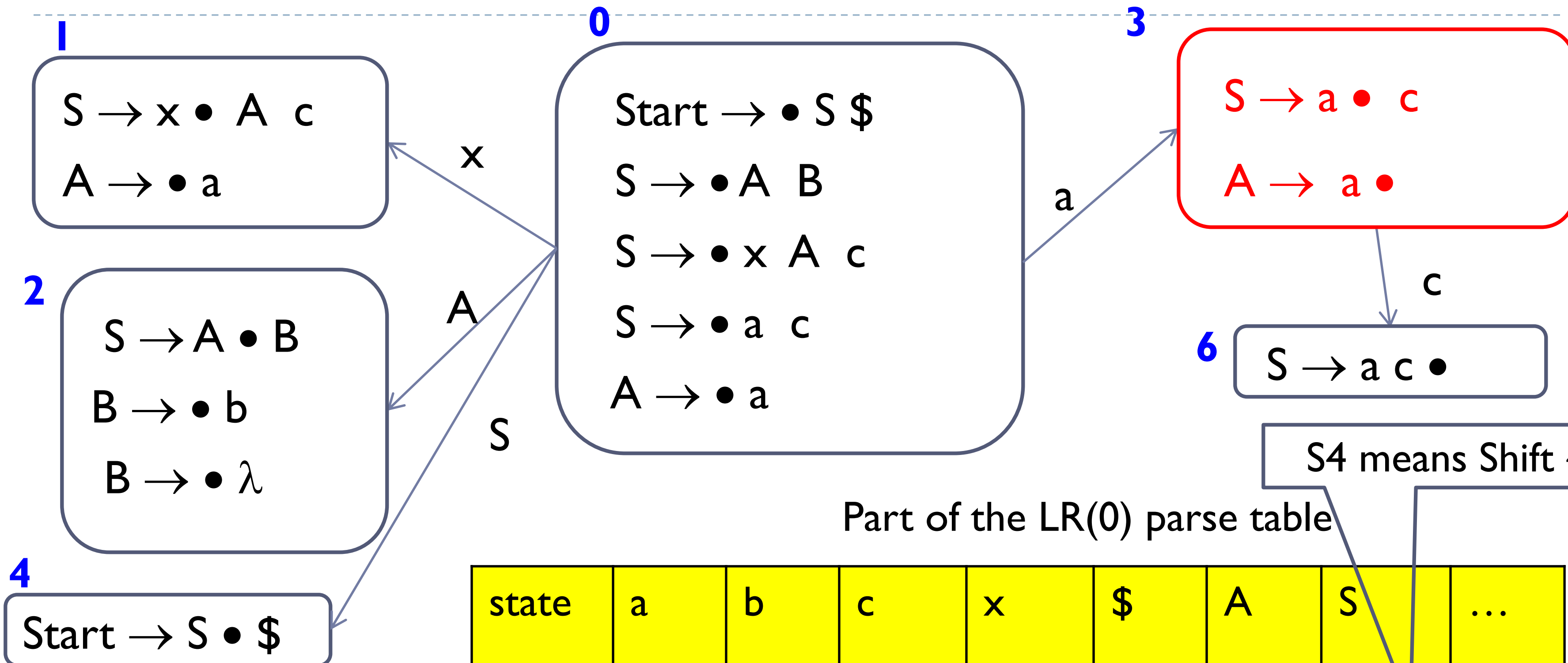
An Example of Using a More Powerful Parser

P_1	Start	$\rightarrow$	S $
P_2	S	$\rightarrow$	A B
P_3		|	a c
P_4		|	x A c
P_5	A	$\rightarrow$	a
P_6	B	$\rightarrow$	b
P_7		|	λ

Its LR(0) Finite Automaton:



Conflict in State 3 in parse table



S4 means Shift 4

Part of the LR(0) parse table

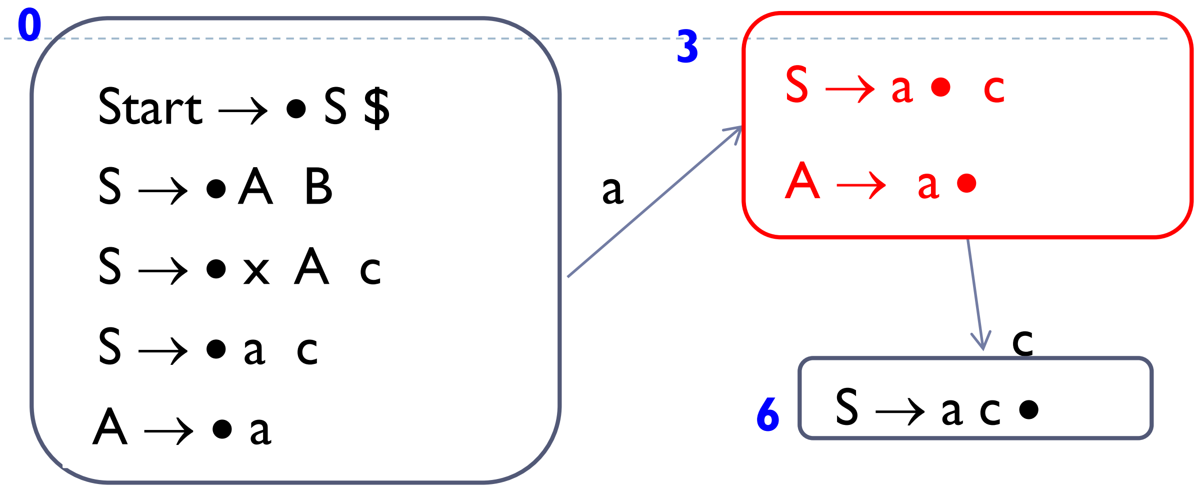
state	a	b	c	x	$	A	S	…
0	S3	err	err	S1	err	S2	S4	…
1								
2								
3	Reduce(A→ a)							

Conflict in parse table[3][c]

When should we do reduce(A → a •) in State 3?

The grammar:

Start → S $
S → A B
| a c
| x A c
A → a
B → b
| λ



Input scenarios:

1) a $
2) a b $
3) a c $

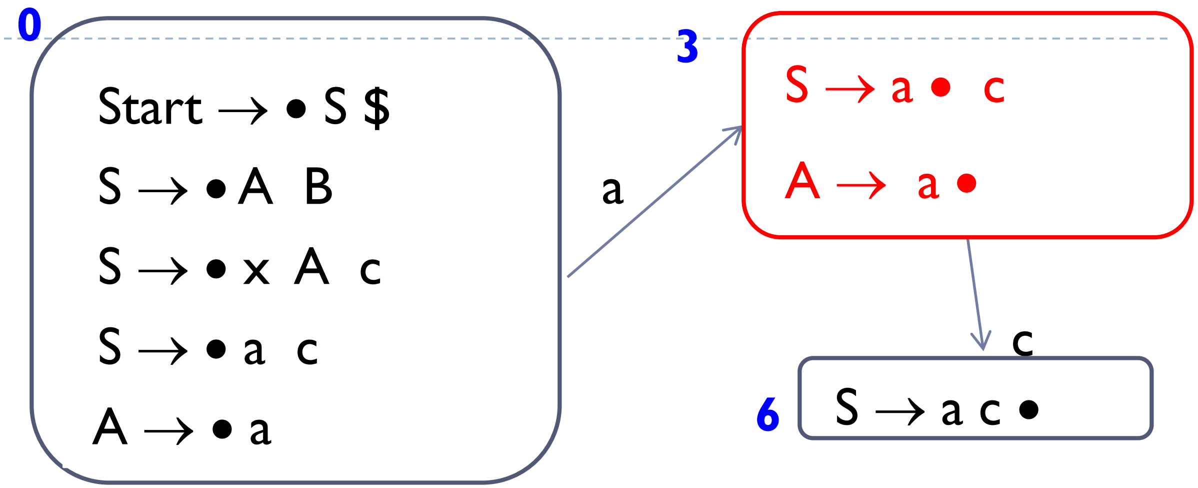
Input scenario 1): a $

i. In state 0, after reading a, we go to state 3.
ii. Next input token is $.
iii. This means S has the structure of A B where A is token a and B is empty.
iv. Therefore, in state 3, if we see $, we should reduce: $ is a token that follows after we do reduce (A → a) in state 3.

When should we do reduce(A → a •) in State 3?

The grammar:

Start → S $
S → A B
  | a c
  | x A c
A → a
B → b
  | λ



Input scenario 2): a b $

i. In state 0, after reading a, we go to state 3.
ii. Next token is b.
iii. This means S has the structure of A B where A is a and B is b.
iv. Therefore, in state 3, if we see b, we should reduce: b is a token that follows after we do reduce (A → a) in state 3.

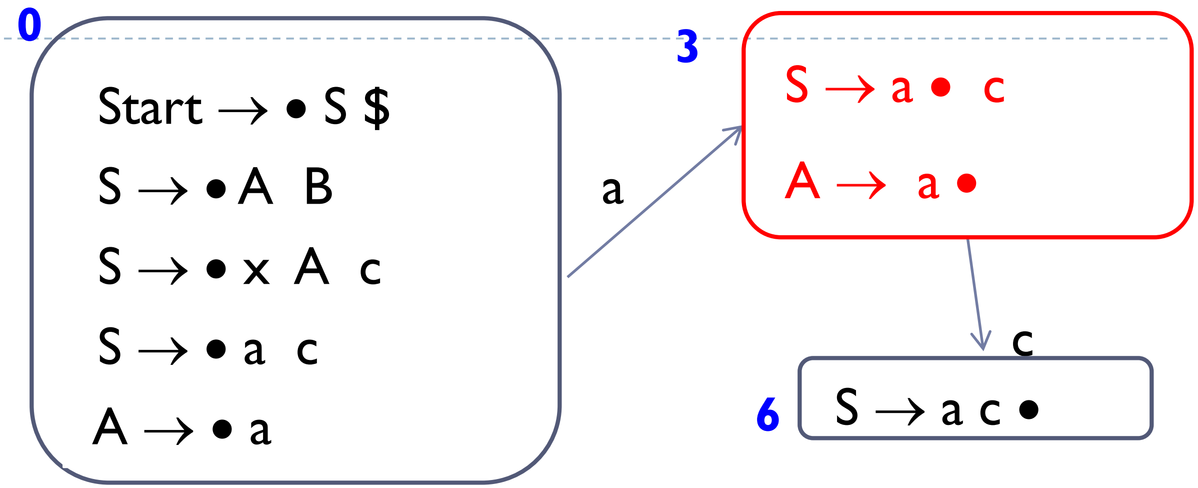
Input scenarios:

1) a $
2) a b $
3) a c $

When should we do reduce(A → a •) in State 3?

The grammar:

Start → S $
S → A B
| a c
| x A c
A → a
B → b
| λ



Input scenario 3): a c $

i. In state 0, after reading a, we go to state 3.
ii. Next token is c.
iii. This means S has the structure of a c .
iv. Therefore, in state 3, if we see c, we should shift to state 6.

Summary: in state 3, if the next token is b or $, we should reduce. We will create a set called itemFollow.

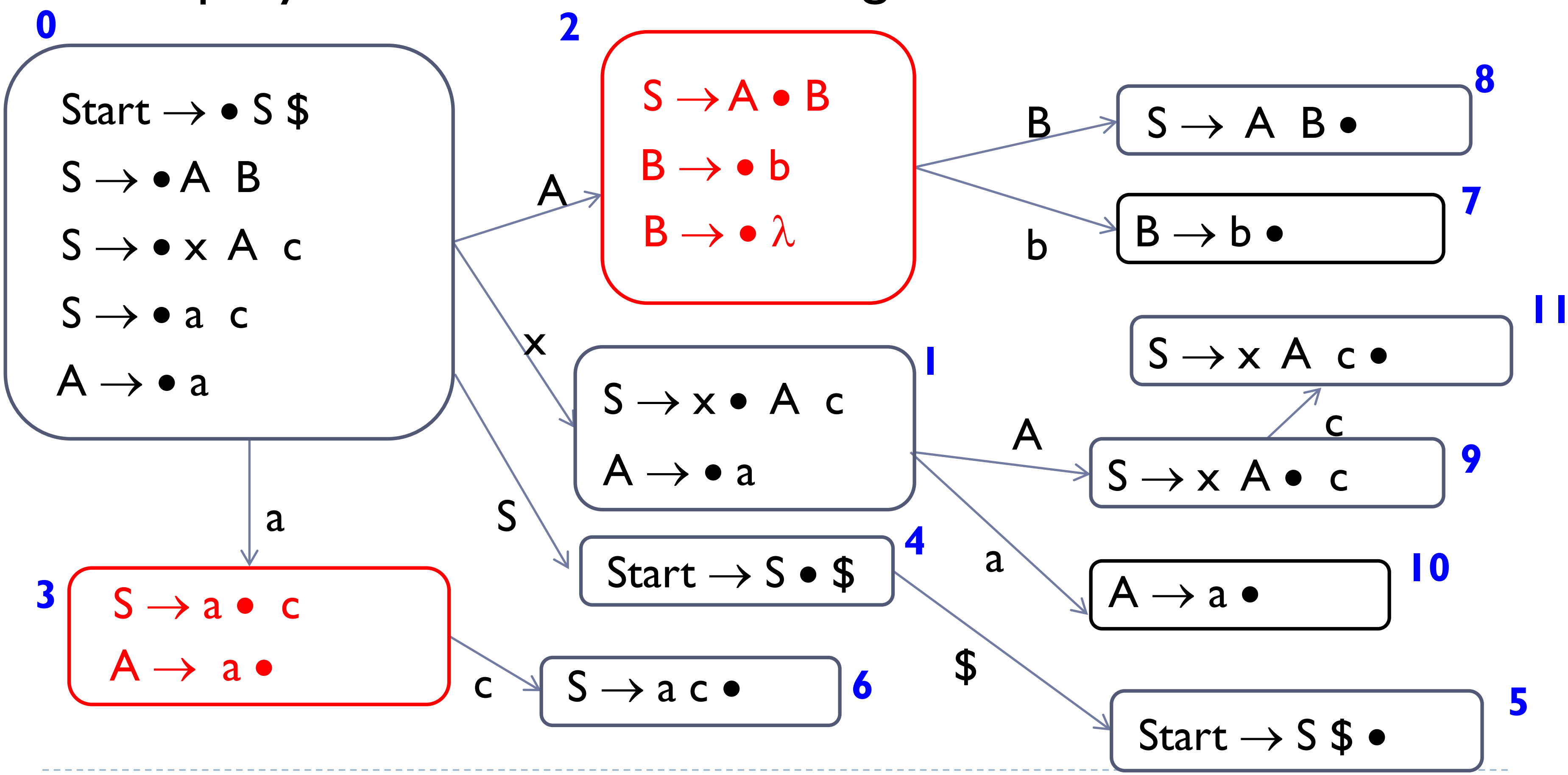
Note this itemFollow set {b, $} is for (3, A → a •). Each item has its own itemFollow set.

LALR(1)—Look Ahead LR with One Token Lookahead

- Due to its balance of power and efficiency, LALR(1) is the most popular LR table-building method.
  - For every transition from state s to state s' labelled with symbol x, enter **shift**(s') into the cell in row s, column x.
  - **If state s contains a final item $A \rightarrow \beta \bullet$, enter reduce($A \rightarrow \beta$) into the cells of row s for each token T $\in$ itemFollow[(*s*, *item*)]**.
  - For cell (0, StartSymbol), enter **accept**.
  - Enter **error** into any remaining empty cells.
- We compute itemFollow[(*s*, *item*)] by using a propagation graph.

LALR Propagation Graph

- Consider an LR(0) transition graph; The pair (*state*, *item*) uniquely identifies each item. E.g.



LALR Propagation Graph

- Each pair (*state*, *item*) is represented by a vertex in the **LALR propagation graph**.
- We will compute itemFollow[ ] for each pair (*state*, *item*).
- The LALR table is constructed with reference to the itemFollow[ ] computed for the items.
- The propagation graph will not be retained after constructing the LALR table.

Generating the Propagation Graph

A. Setup:

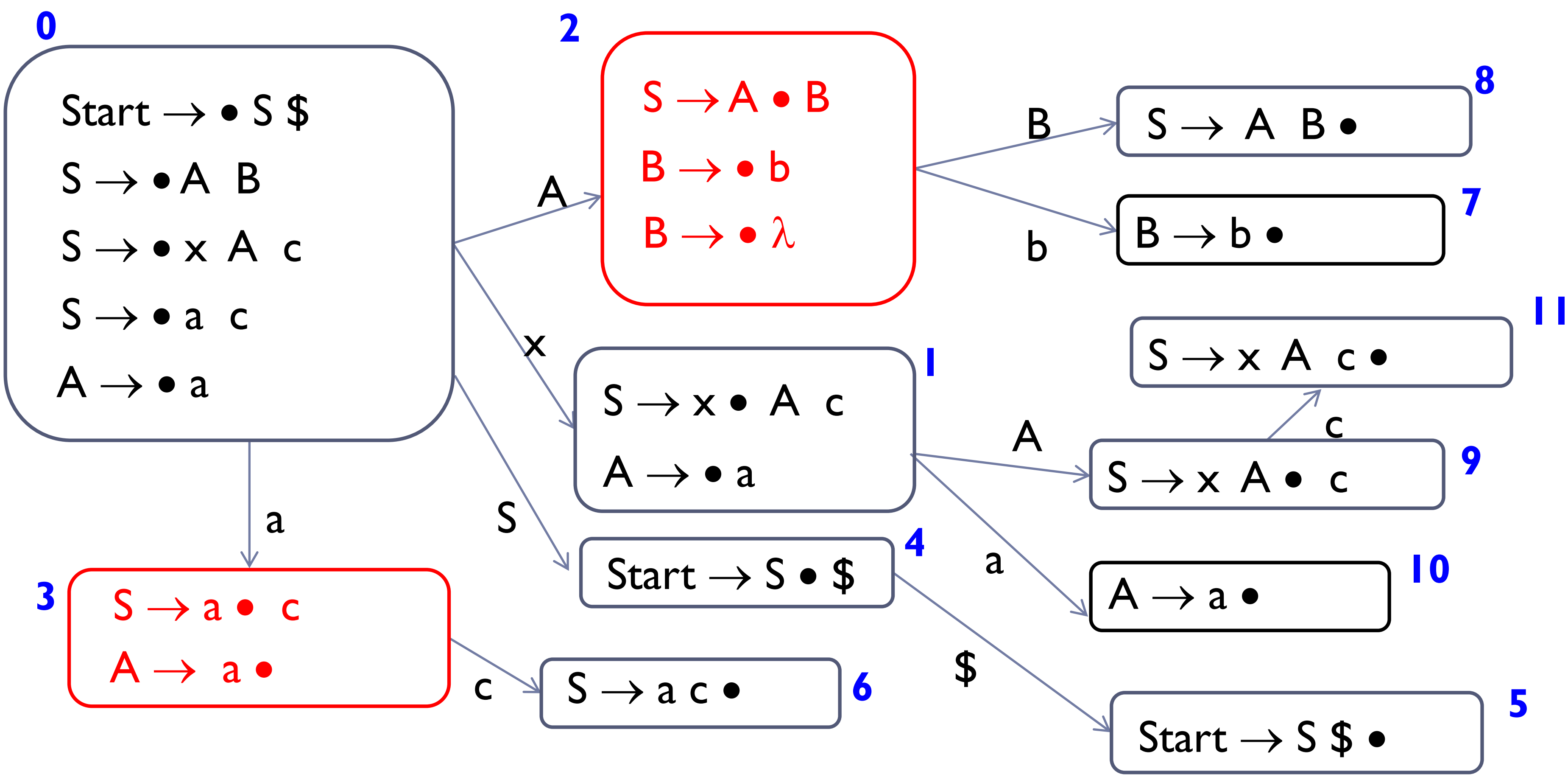
   1. Create the LR(0) finite automaton.
   2. For each (state, item), create a vertex *v* in the graph.
   3. Initialise all itemFollow[*v*] to ∅.
   4. Initialise itemFollow[(0, StartSymbol Productions)] = {$}

B. Build the propagation graph

C. Propagate itemFollow[ ]

Note, an array itemFollow is used to store the itemFollow set for each vertex v in the propagation graph.

A: Setup (create the LR(0) finite automaton)



A: Setup (create vertices, initialise itemFollow)

Vertex no.	state	itemFollow
1	0, Start $\rightarrow \bullet$ S \$	{\$}
2	0, S $\rightarrow \bullet$ A B	$\emptyset$
3	0, S $\rightarrow \bullet$ x A c	$\emptyset$
4	0, S $\rightarrow \bullet$ a c	$\emptyset$
5	0, A $\rightarrow \bullet$ a	$\emptyset$
6	1, S $\rightarrow$ x $\bullet$ A c	$\emptyset$
7	1, A $\rightarrow \bullet$ a	$\emptyset$
8	2, S $\rightarrow$ A $\bullet$ B	$\emptyset$
9	2, B $\rightarrow \bullet$ b	$\emptyset$
10	2, B $\rightarrow \bullet\ \lambda$	$\emptyset$
11	3, S $\rightarrow$ a $\bullet$ c	$\emptyset$
12	3, A $\rightarrow$ a $\bullet$	$\emptyset$
13	4, Start $\rightarrow$ S $\bullet$ \$	$\emptyset$
14	5, Start $\rightarrow$ S \$ $\bullet$	$\emptyset$
15	6, S $\rightarrow$ a c $\bullet$	$\emptyset$
16	7, B $\rightarrow$ b $\bullet$	$\emptyset$
17	8, S $\rightarrow$ A B $\bullet$	$\emptyset$
18	9, S $\rightarrow$ x A $\bullet$ c	$\emptyset$
19	10, A $\rightarrow$ a $\bullet$	$\emptyset$
20	11, S $\rightarrow$ x A c $\bullet$	$\emptyset$

B: Building the Propagation Graph (ideas)

1) For each pair $(s, A \rightarrow \alpha \bullet B\gamma)$, put first($\gamma$) into the itemFollow set of $(s, B \rightarrow \bullet\delta)$.

   The result is the token(s) that follow B is recorded in the itemFollow set of the initial item of B.

2) For each pair $(s, A \rightarrow \alpha \bullet X\gamma)$, a propagation edge is placed from $(s, A \rightarrow \alpha \bullet X\gamma)$ to $(t, A \rightarrow \alpha X \bullet \gamma)$.

   The result is, for each initial item in each state, we build a path that starts from the initial item to reach its final item.

3) For each pair $(s, A \rightarrow \alpha \bullet B\gamma)$, when $\gamma =>^* \lambda$, place a propagation edge from $(s, A \rightarrow \alpha \bullet B\gamma)$ to $(s, B \rightarrow \bullet\delta)$.

   The result is, since γ may be empty, we tell the initial item of B that the tokens that may follow B may come from the tokens following A

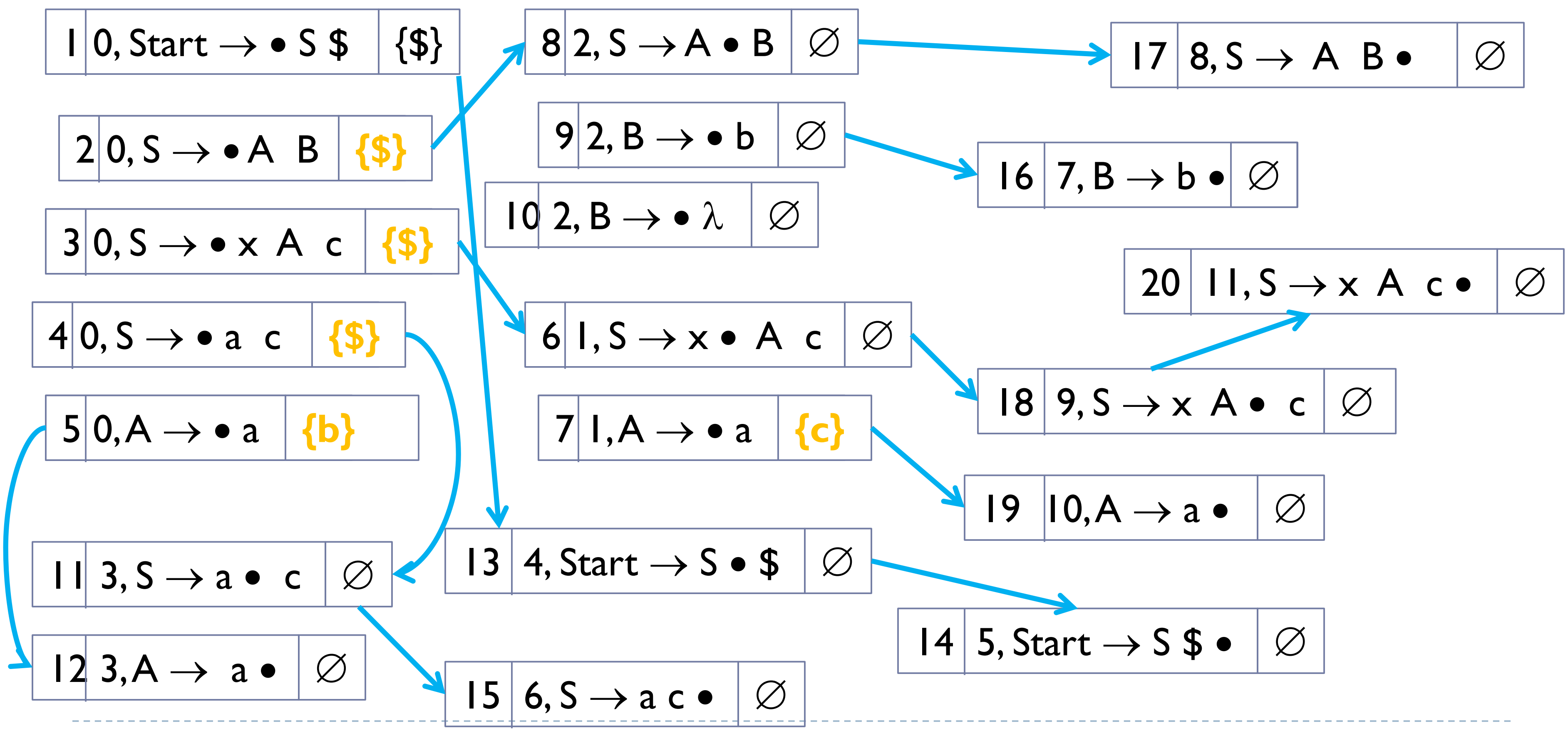
B: Building the Propagation Graph

~~For each pair (s, A $\rightarrow$ $\alpha\bullet B\gamma$), put first(γ) into the itemFollow set of (s, B $\rightarrow$ $\bullet\delta$)~~

1	0, Start $\rightarrow$ • S \$	{\$}
2	0, S $\rightarrow$ • A B	{\$}
3	0, S $\rightarrow$ • x A c	{\$}
4	0, S $\rightarrow$ • a c	{\$}
5	0, A $\rightarrow$ • a	{b}
6	1, S $\rightarrow$ x • A c	$\varnothing$
7	1, A $\rightarrow$ • a	{c}
8	2, S $\rightarrow$ A • B	$\varnothing$
9	2, B $\rightarrow$ • b	$\varnothing$
10	2, B $\rightarrow$ • λ	$\varnothing$
11	3, S $\rightarrow$ a • c	$\varnothing$
12	3, A $\rightarrow$ a •	$\varnothing$
13	4, Start $\rightarrow$ S • \$	$\varnothing$
14	5, Start $\rightarrow$ S \$ •	$\varnothing$
15	6, S $\rightarrow$ a c •	$\varnothing$
16	7, B $\rightarrow$ b •	$\varnothing$
17	8, S $\rightarrow$ A B •	$\varnothing$
18	9, S $\rightarrow$ x A • c	$\varnothing$
19	10, A $\rightarrow$ a •	$\varnothing$
20	11, S $\rightarrow$ x A c •	$\varnothing$

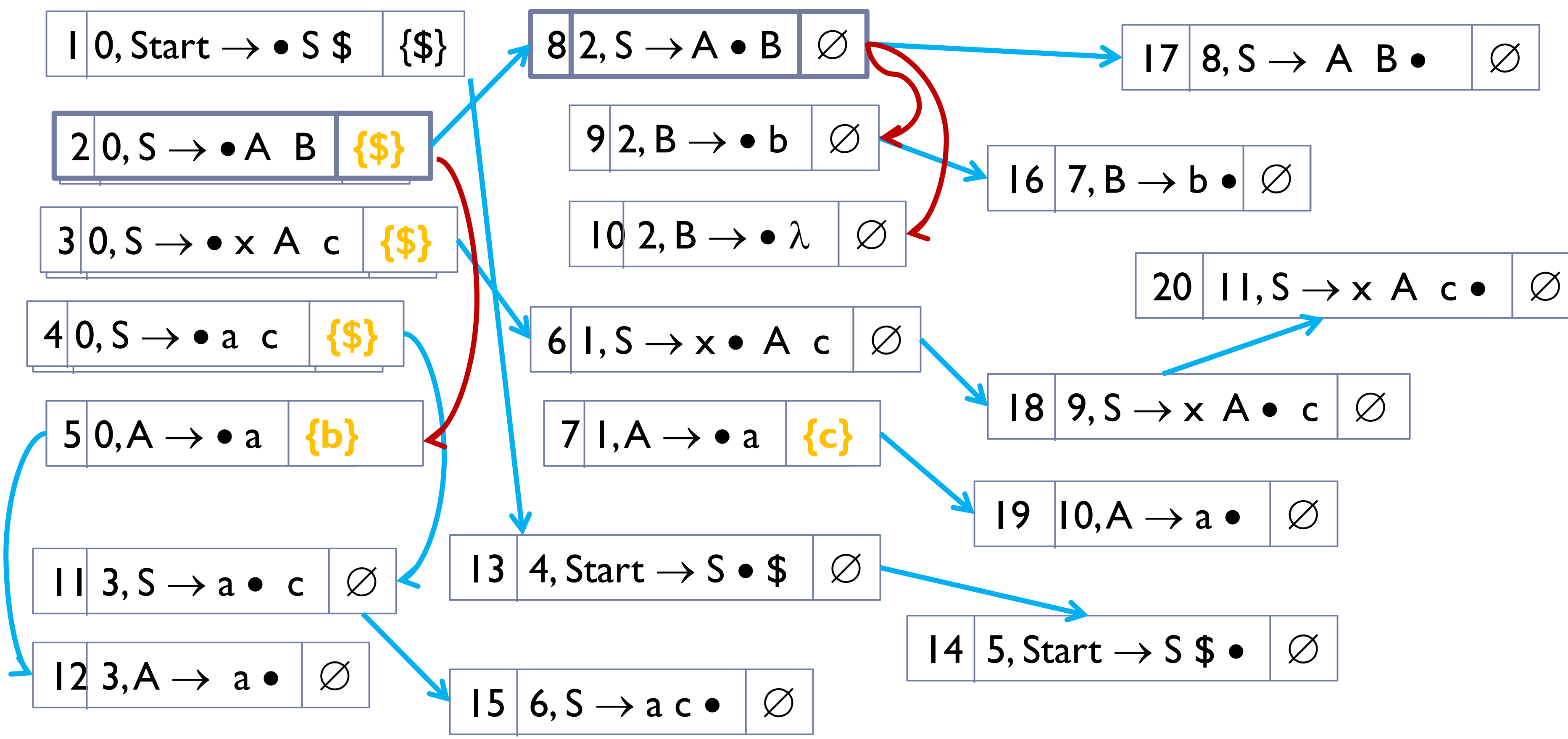
B: Building the Propagation Graph

For each pair (s, A → α•Xγ), a propagation edge is placed from (s, A → α•Xγ) to (t, A → αX•γ)



B: Building the Propagation Graph

For each pair (s, A → α•Bγ), when γ=>*λ, place a propagation edge from (s, A → α•Bγ) to (s, B → •δ)

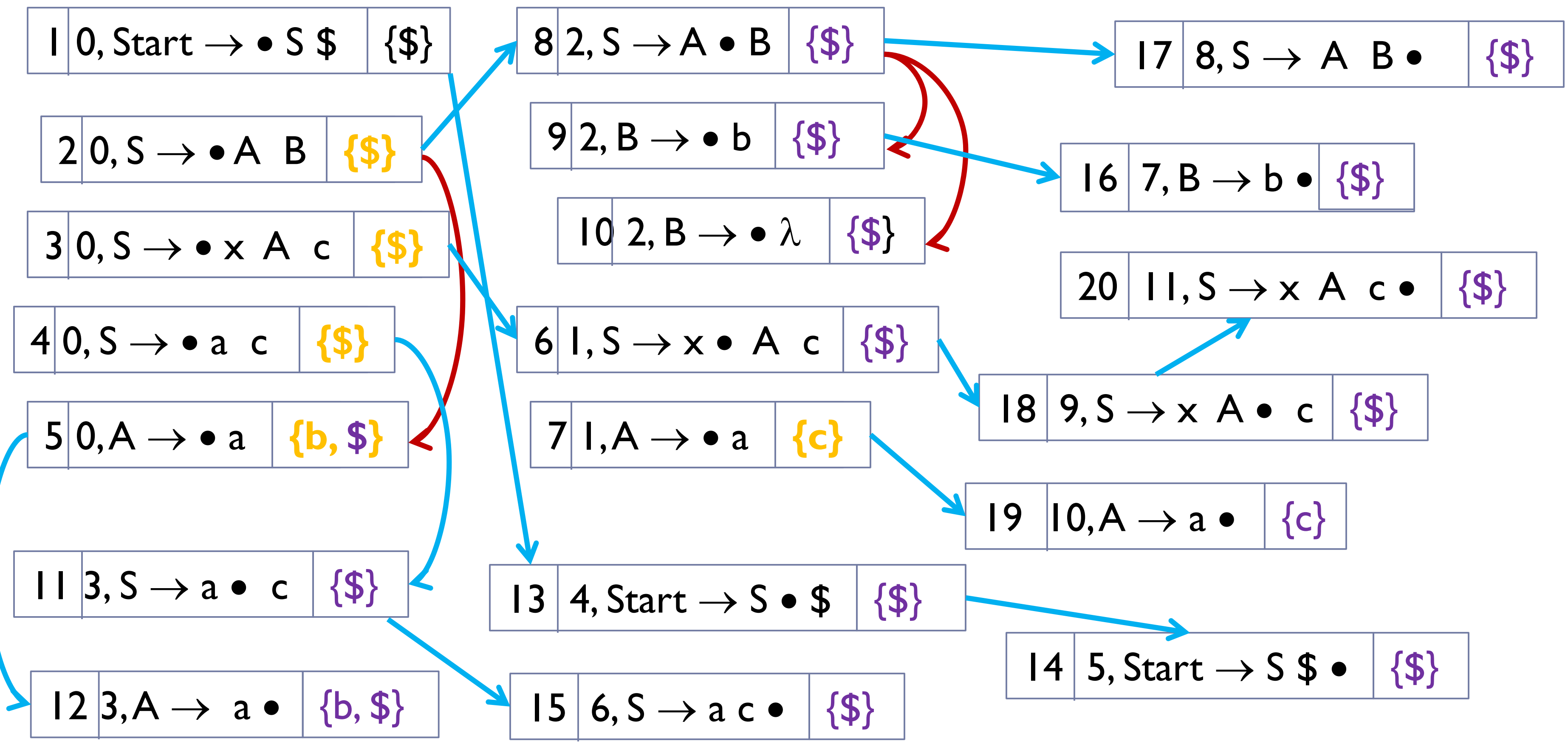


C: Propagating itemFollw [ ]

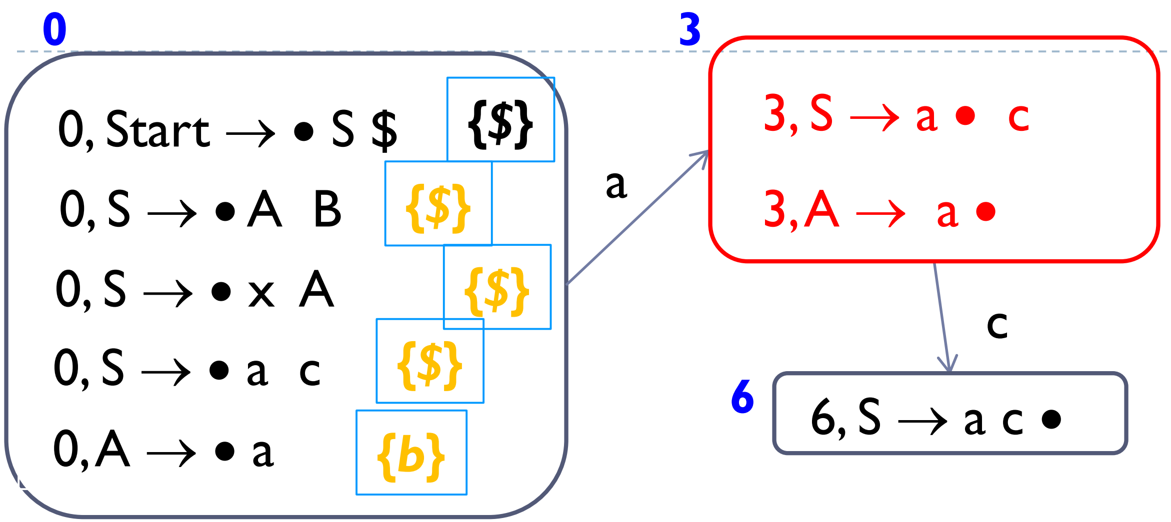
While (making progress)

  For each edge (*u*,*v*)

    add ItemFollow[*u*] to ItemFollow[*v*]



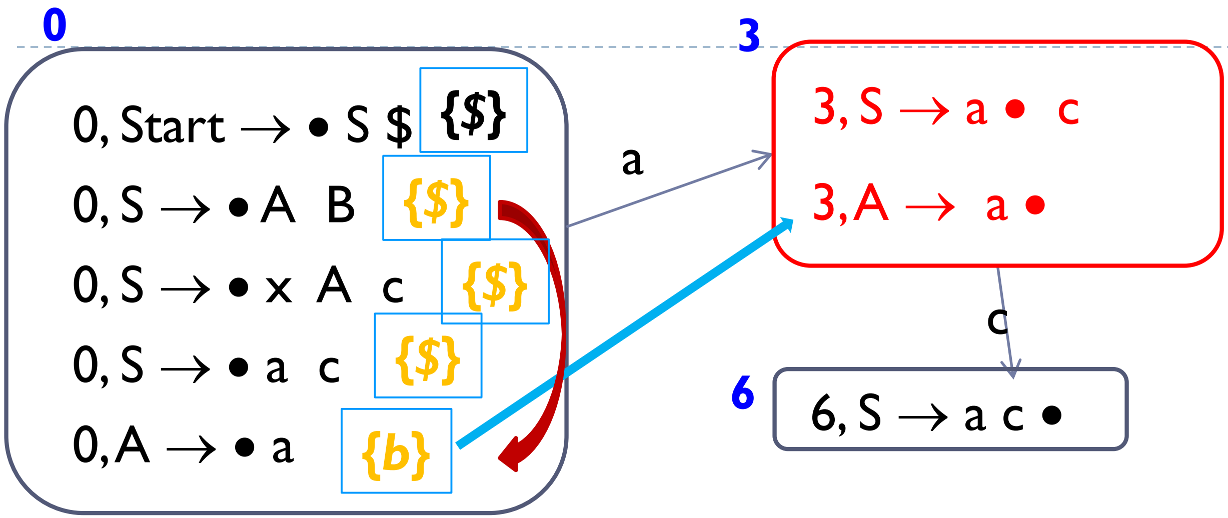
Explanations Using the Example



i. itemFollow[(0, Start → • S $)] initialised to {$}. Explanation: $ follows the Start symbol.

ii. For (0, Start → • S $), put first($) into the itemFollow sets of the 3 initial items of S in state 0. Explanation: $ follows S. This info will propagate through the edges from the initial items to the final items of S in other states. Similarly, for (0, S → • A B), put first(B) into …

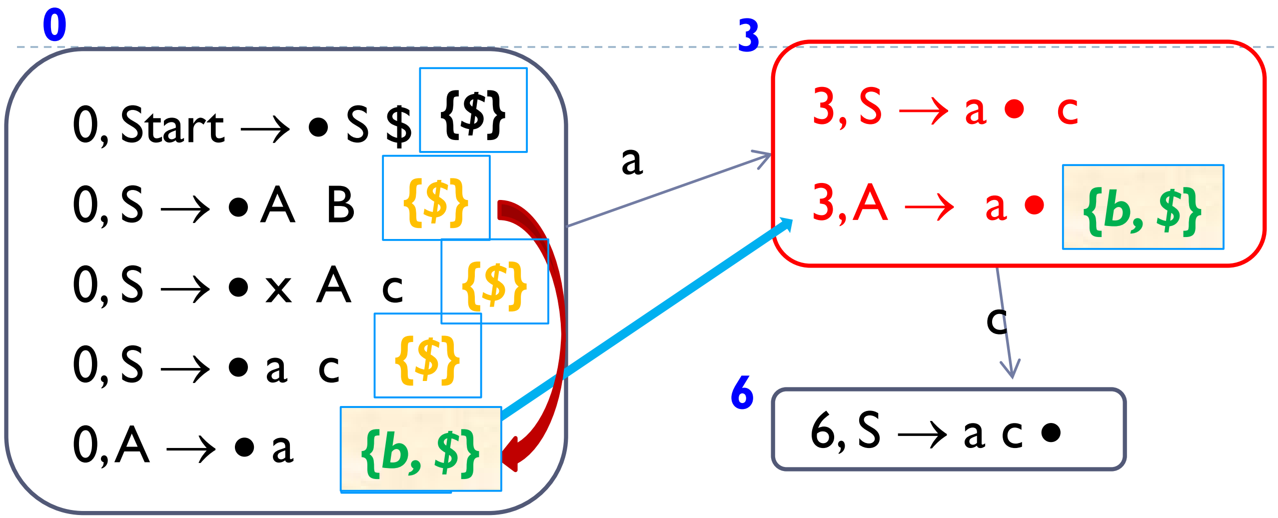
Explanations Using the Example



iii. Put a propagation edge from (0, A → • a) to (3, A → a •). Explanation: To facilitate propagation of itemFollow from the initial item to the final item of A.

iv. Put a propagation edge from (0, S → • A B) to (0, A → • a) because B may be empty. Explanation: When B is empty, tokens following S follow A. So itemFollow of S should be passed/propagated to itemFollow of A.

Explanations Using the Example



v. Propagate itemFollow contents through propagation edges

The Result:

The LALR(1) parse table: **itemFollow[(3, A → a •)] = {b, $}**

state	…	b	c	$	…
…	…				
3		Reduce(A→ a)	Shift 6	Reduce(A→ a)	
…	…				

Compare with the LR(0) parse table in slide 70:

state	a	b	c	x	$	A	S	…
…	…							
3	Reduce(A→ a)							
…	…							

LALR(1) Parser

- LALR(1) grammars are available for all popular programming languages.
- LALR(1) is thus a powerful parsing method that can handle all popular programming languages.

Parser Generator Beaver

- Beaver is a LALR(1) parser generator that generates parsers written in Java.
- LALR(1) parsers can accept grammar rules with left recursions and common prefixes.

Parser Generator Beaver

- Beaver's syntax is very similar to the notation we have been using for context-free grammars, except that Beaver uses = where we have used $\rightarrow$.
- The rules for a nonterminal must be terminated by a semicolon.
- The directive %**terminals** on the first line declares the set of terminals used by the grammar.
- The directive %**goal** specifies the start symbol.
- Beaver implicitly assumes that every name that isn't declared to be a terminal is a nonterminal.

Example of a Beaver Specification: Grammar for Arithmetic Expressions

```
%terminals PLUS, MINUS, MUL, DIV, NUMBER, LPAREN, RPAREN;
%goal Expr;
Expr = Expr PLUS Term
     | Expr MINUS Term
     | Term
     ;
Term = Term MUL Factor
     | Term DIV Factor
     | Factor
     ;
Factor = NUMBER
     | LPAREN Expr RPAREN
     ;
```

Compare how we write in slide 8:

Expr	$\rightarrow$	Expr plus Term
	|	Expr minus Term
	|	Term
Term	$\rightarrow$	Term mul Factor
	|	Term div Factor
	|	Factor
Factor	$\rightarrow$	number
	|	id
	|	lparen Expr rparen

Test Yourself 3.4

1. What does a syntax analyser do?
2. What are the input and output of a syntax analyser?
3. What is a context-free grammar used for in a compiler?
4. Write the pseudocode for a recursive descent parser for the context-free grammar of Expr on slide 8 (reference slide 20). Assume the methods peek(), match() and predict() are provided.
5. Follow the bottom-up parsing engine on slide 48 and use the parse table on slide 49 for the grammar on slide 46 to trace the parsing process for the input "adc$".

Test Yourself 3.4

6. Consider the following grammar rules where Stmt, Expr, IndexExpr, and TypeName are non-terminal symbols and ';', ID, '[', and ']' are tokens. Stmt is the start symbol. This question is better done after the tutorial class on this topic has been completed.

Stmt → Expr ';'
  | TypeName ID ';'
Expr → ID
  | IndexExpr
IndexExpr → ID '[' Expr ']'
  | IndexExpr '[' Expr ']'
TypeName → ID
  | TypeName '[' ']'

Test Yourself 3.4

1) Construct state 0 and state 1 of the LR(0) automaton. State 0 is the start state, and state 1 is reached after reading ID in state 0. Identify the reduce/reduce and shift/reduce conflicts in state 1.

2) Build the part of the LALR(1) propagation graph to show only the items in state 0 and state 1 of the LR(0) automaton, and compute the *itemFollow* sets for these items. Show that the reduce/reduce conflict is gone but the shift/reduce conflict still exists.

Test Yourself 3.1 (answers)

$$
\begin{aligned}
BExpr &\rightarrow BExpr\ OR\ BTerm \\
&\mid BExpr\ XOR\ BTerm \\
&\mid BTerm \\[1em]
BTerm &\rightarrow BTerm\ AND\ BFactor \\
&\mid BFactor \\[1em]
BFactor &\rightarrow ZERO \\
&\mid ONE \\
&\mid LPAREN\ BExpr\ RPAREN \\
&\mid COMPLEMENT\ BFactor
\end{aligned}
$$

Test Yourself 3.1 (answers)

Explanations

Please refer to slide 15 where we put the div and add operators into the rules for Expr, then we have ambiguous grammar. So if we have

BExpr -> BExpr AND BExpr

| BExpr OR BExpr

| BExpr XOR BExpr

we will also have ambiguous grammar. For example, bit expressions like "0 OR 1 AND 1" can result in two different parse trees from leftmost derivation. One of these two trees will be the wrong order (or is done before and),

Test Yourself 3.1 (answers)

For the same reason, if we have

BExpr -> BExpr AND BExpr

  | BExpr OR BExpr

  | BExpr XOR BExpr

  | COMPLEMENT BExpr

we will also have ambiguous grammar for bit expressions like ”~1 AND 1”. You can also draw two different parse trees from leftmost derivation and one tree will do AND before ~.

Therefore, we have to put operators that are at the same level of precedence in the rules for one nonterminal symbol. So OR and XOR are in the rules for BExpr, AND is in BTerm, and COMPLEMENT is in BFactor.

Test Yourself 3.2 (answers) – Q1

Start by computing first(A B c)—find the first tokens of "A B c":

Ans = {a}   after considering rule 2;

No change after considering rule 3;

Since A may be empty, find the first tokens of "B c":

Ans = {a, b} from rule 4;

No change after considering rule 5;

Since B may be empty, find the first tokens of "c":

Ans = {a, b, c};

Since rule 1, i.e. "A B c" cannot be empty, we do not need to compute follow(S);

Therefore, predict(S $\rightarrow$ A B c) = {a, b, c}.

Test Yourself 3.2 (answers) – Q2

Start by computing first(λ)—find the first tokens of λ:

Ans = $\varnothing$;

Since it is an empty rule, we need to compute follow(A)—find what tokens may follow A:

Find all occurrences of A in the RHS of all rules—there is only one occurrence of A—in rule 1;

Find first(B c) which re sturns {b, c};

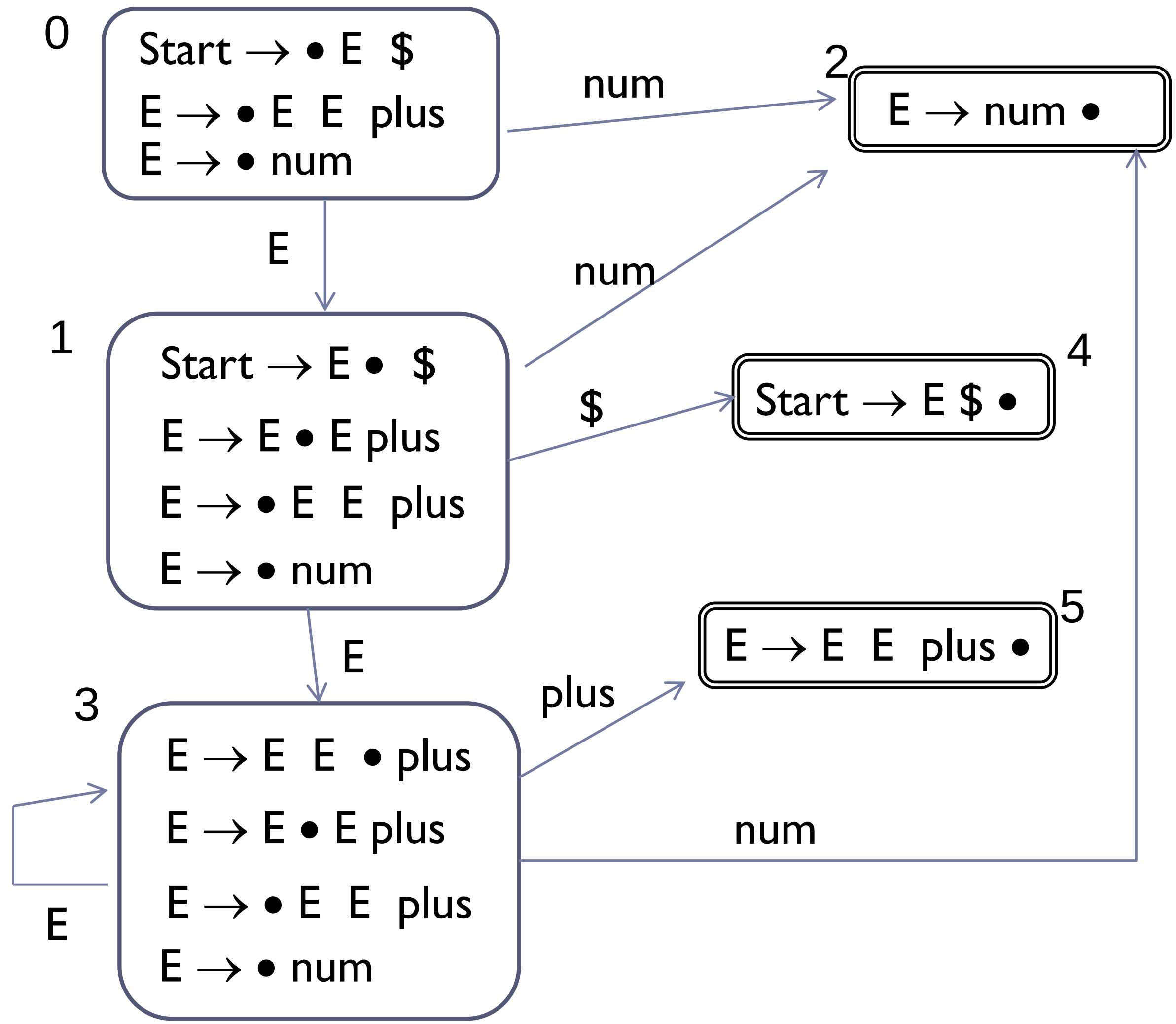
Therefore follow(A) = {b, c};

Therefore predict($A \rightarrow \lambda$) = {b, c}.

Similarly, predict($B \rightarrow \lambda$) = {c}.

Test Yourself 3.3 (answers)

LR(0) transition graph



Test Yourself 3.3 (answers)

LR(0) parse table

state	num	plus	$	Start	E
0	Shift 2	error	error	accept	Shift 1
1	Shift 2	error	Shift 4	error	Shift 3
2	Reduce (E → num)				
3	Shift 2	Shift 5	error	error	Shift 3
4	Reduce (Start → E $)				
5	Reduce(E → E E plus)				

Test Yourself 3.4 (Q6)

0

Stmt → • Expr ';'
Stmt → • TypeName ID ';'
Expr → • ID
Expr → • IndexExpr
IndexExpr → • ID '[' Expr ']'
IndexExpr → • IndexExpr '[' Expr ']'
TypeName → • ID
TypeName → • TypeName '[' ']'

↓ **ID**

1

Expr → ID •
IndexExpr → ID • '[' Expr ']'
TypeName → ID •

Reduce/reduce conflict between the two final items and shift/reduce conflict between final items and non-final item.

Test Yourself 3.4 (Q6)

Each row is one vertex in the propagation graph.

1	0, Stmt → • Expr ';'	{$}
2	0, Stmt → • TypeName ID ';'	{$}
3	0, Expr → • ID	{';'}
4	0, Expr → • IndexExpr	{';'}
5	0, IndexExpr → • ID '[' Expr ']'	{'[', ';'}
6	0, IndexExpr → • IndexExpr '[' Expr ']'	{'[', ';'}
7	0, TypeName → • ID	{ID, '['}
8	0, TypeName → • TypeName '[' ']'	{ID, '['}
9	1, Expr → ID •	{';'}
10	1, IndexExpr → ID • '[' Expr ']'	{';', '['}
11	1, TypeName → ID •	{ID, '['}

No reduce/reduce conflict in state 1

But shift/reduce conflict between vertex 10 and vertex 11

Acknowledgements

Slides 49–55: Fischer, Cytron and LeBlanc, Jr. (2010). *Crafting a Compiler. 1st Edition*. Boston: Pearson Education, Inc.