Compiler Techniques

3. Syntax Analysis

Huang Shell Ying

Overview

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

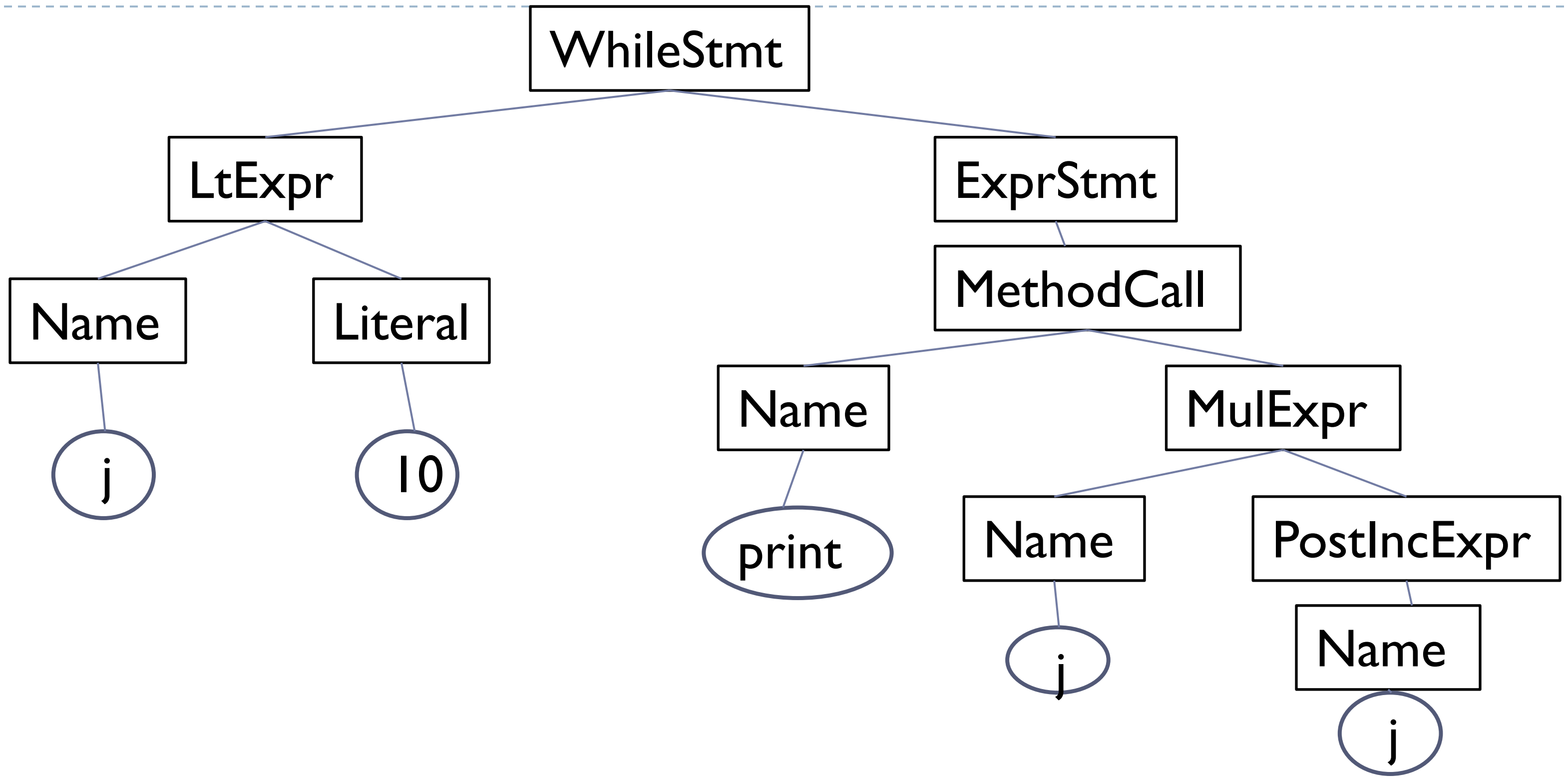
For example,

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

WHILE	LPAR	ID	LT	LIT	RPAR	ID	LPAR	ID	MUL	LPAR	…

back

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

- 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 meaningful 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 syntax can be described by a **context-free grammar** (can we use regular expressions?).
- A context-free grammar $G = (N, T, P, S)$ 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 $A \to s_1 \ldots s_n$ 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 $A \to \lambda$.

Example of a context free grammar

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

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

Factor → number
Factor → id
Factor →lparen *Expr* rparen

back

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	$(N \cup T)^*$

back

Deriving Sentences

- The language defined by a grammar is <u>the set of all sentences</u> that can be derived from its start symbol.
- Example of one sentence derived using the rules on slide 6:

Expr ⇒ *Expr* plus *Term*
⇒ *Term* plus *Term*
⇒ *Factor* plus *Term*
⇒ *id* plus *Term*
⇒ *id* plus *Term* mul *Factor*
⇒ *id* plus *Factor* mul *Factor*
⇒ *id* plus *id* mul *Factor*
⇒ *id* plus *id* mul *id*

- Since there may be multiple rules for a nonterminal, derivation is non-deterministic: we may derive many different sentences from the same initial phrase. (try to derive 3 more sentences for the Expr language)

Deriving Sentences

- If $A \rightarrow \beta$ is a rule of G and $\alpha, \gamma \in (N \cup T)^*$ are phrases of G, then $\alpha A\gamma \Rightarrow \alpha\beta\gamma$ is a **one-step derivation**

  E.g. ***Expr*** plus ***Term*** $\Rightarrow$ ***Expr*** minus ***Term*** plus ***Term***

Slide 6

- $\alpha \Rightarrow^{+} \beta$ means that β is derived in ***one or more*** steps from α,

  $\alpha \Rightarrow^{*} \beta$ in ***zero or more*** steps

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

Derivation Sequences

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

  $\textbf{\textit{Term}} \Rightarrow \textbf{\textit{Term}}\ \text{mul}\ \textbf{\textit{Term}}$

- 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.

$$
\begin{array}{lll}
\textit{Term} & \rightarrow & \textit{Term}\ \text{mul}\ \textit{Factor} \\
 & | & \textit{Term}\ \text{div}\ \textit{Factor} \\
 & | & \textit{Factor}
\end{array}
$$

Parse Trees (this is not the AST)

- A ***parse tree*** **represents a derivation**, but abstracts away from the order of derivations.

- 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.

- The labelling obeys the following requirement:

  A node labelled with A which has children labelled $s_1 \dots s_n$, if and only if there is a rule $A \rightarrow s_1 \dots s_n$

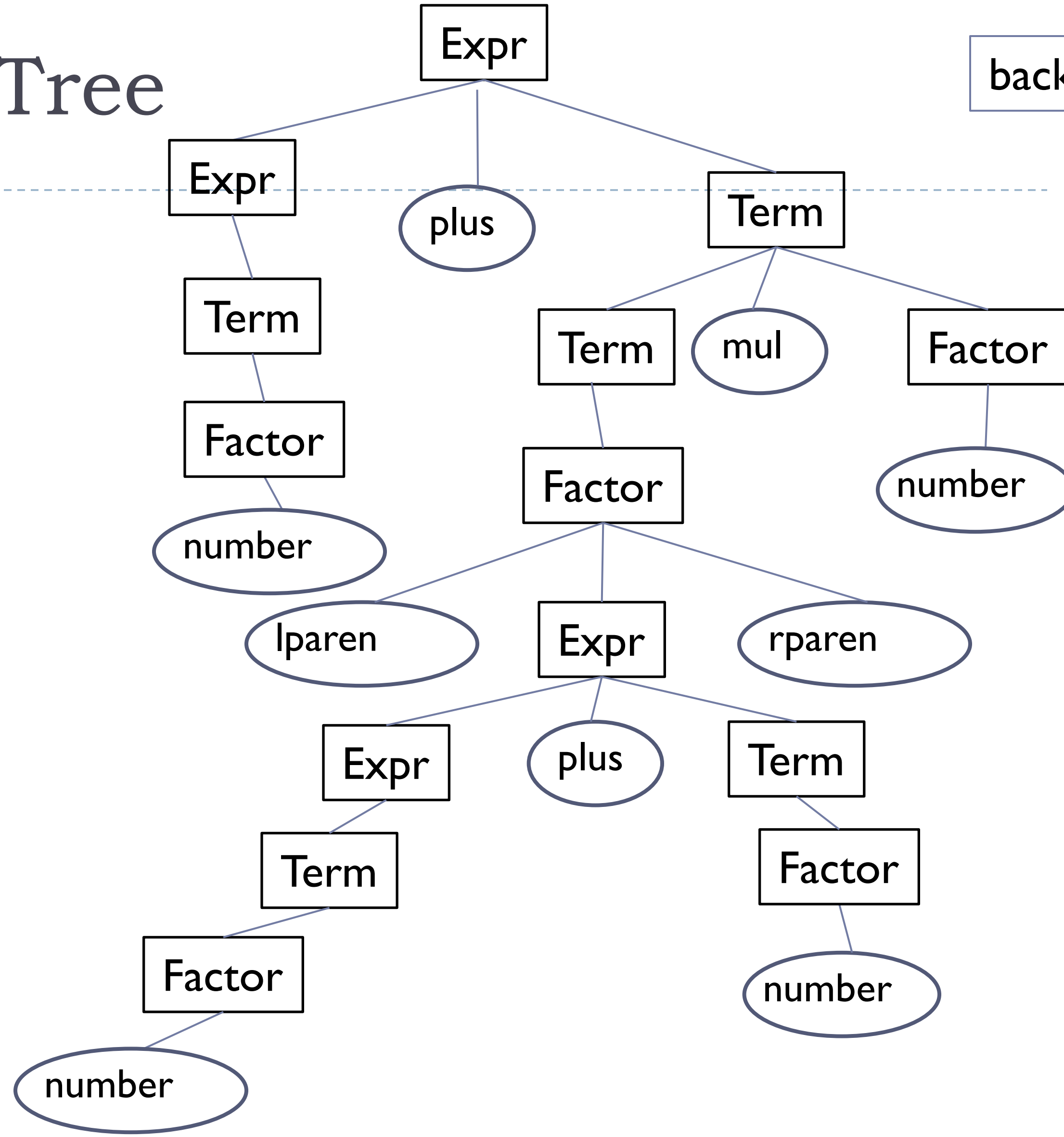
Example Parse Tree

back

Expr → *Expr* plus *Term*
| *Expr* minus *Term*
| *Term*

Term → *Term* mul *Factor*
| *Term* div *Factor*
| *Factor*

Factor → number
| lparen *Expr* rparen



Sentence derived:

number plus lparen number plus number rparen mul number

Ambiguous Grammars

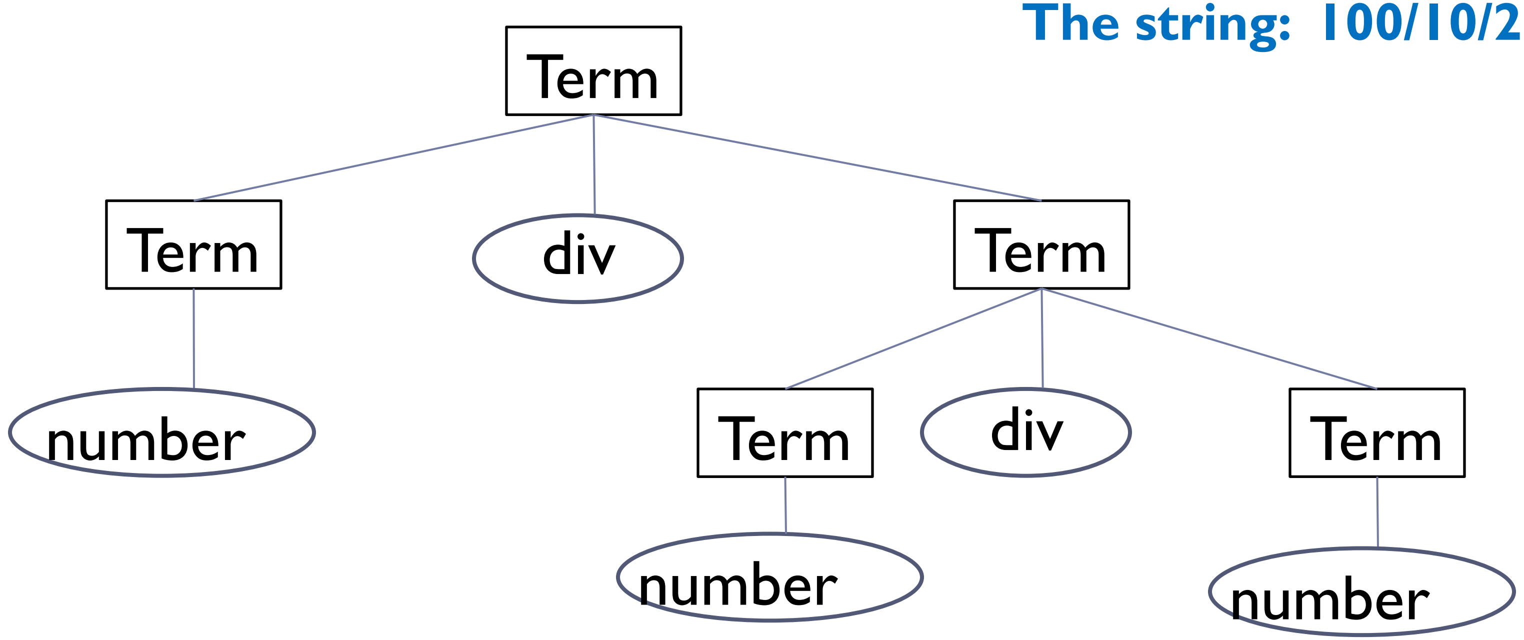
- Consider the grammar *G* defined by the following two rules:

Term $\rightarrow$ Term div Term // rule 1

| number // rule 2

A parse tree for number div number div number

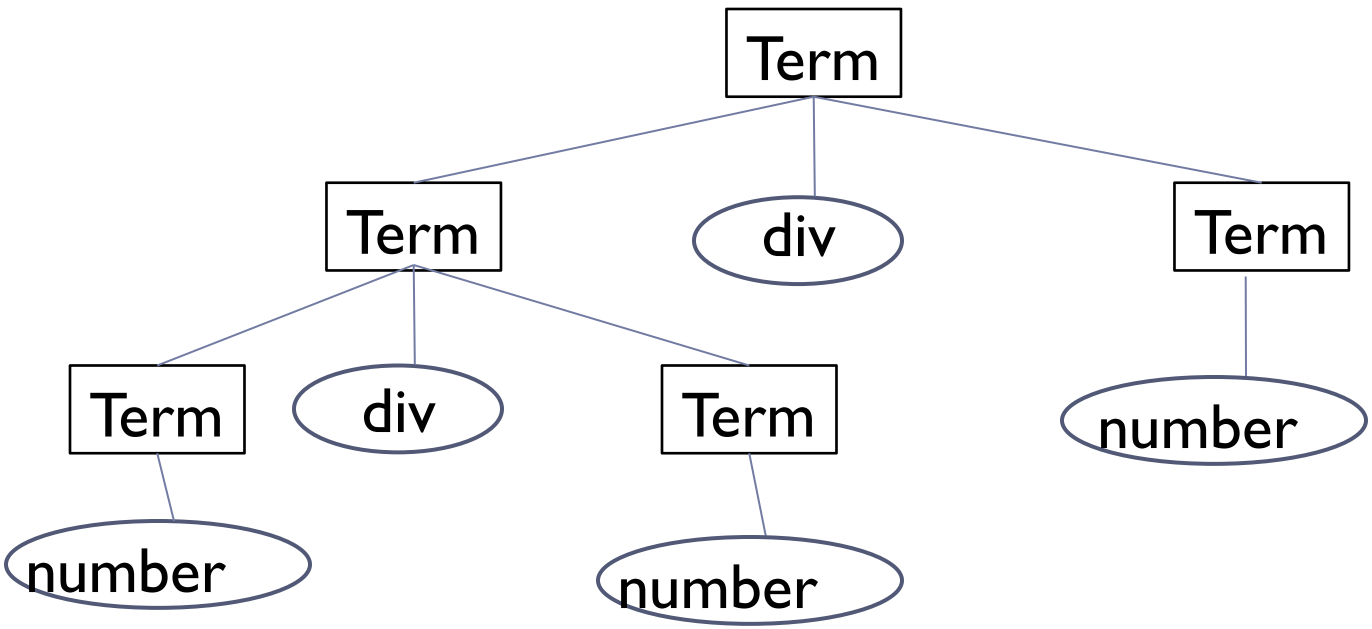
The string: 100/10/2



Ambiguous Grammars

Another parse tree for
number div number div number

The string: 100/10/2



Ambiguous Grammars

- **A grammar that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence are called ambiguous.**
- Ambiguous grammars are not useful for compilers.
- There is no algorithm that can check an arbitrary context free grammar for ambiguity.

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 $w \in L(G)$.

Slide 2

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

Slide 12

- ***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.

back

Top-down Parsing

back

- 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.

Slide 12

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

back

- 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 {…}

Recursive Descent Parsing

- If there is more than one rule for A, parseA inspects the next input token(s) and choose a production rule among the rules for A to apply.

Slide 12

- The code for applying a production rule is obtained by processing the RHS of the rule, symbol by symbol

  $A \rightarrow X_1X_2\ldots X_m$

Slide 21

- If the next symbol X_i is a terminal t, confirms 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.

back

Recursive Descent Parsing

back

- The parsing of the whole program starts from the parse method for the start symbol.
- Recursive descent parsers use one token of lookahead to determine which rule to use.
- Lookahead has to be unambiguous: there cannot be more than one rule (for the same nonterminal) whose RHS starts with the same token.

Slide 21

- 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***.

Example

Slide 17 back

```
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
$\mid\ \lambda$	p_3
$A \to a\ B\ C\ d$	p_4
$\mid\ B\ Q$	p_5
$B \to b\ B$	p_6
$\mid\ \lambda$	p_7
$Q \to q$	p_8
$\mid\ \lambda$	p_9

/* match() confirms a token */

Computing predict(p)

back

Consider a production rule p: $X \to X_1 X_2 \ldots X_m$, $m \geq 0$.

Slide 7

- The set of tokens that **predicts** rule p includes
  - The set of possible tokens that are first produced in some derivation from $X_1 X_2 \ldots X_m$
    - 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.
  - Those tokens that can follow X in some derivation from $X_1 X_2 \ldots X_m$
    - If $X_1 X_2 \ldots X_m$ may be empty, the first tokens that may follow X

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

Computing predict(p)

Example: p: **$A \rightarrow B\ Q$**

- The set of possible tokens that are first produced in some derivation from B Q for rule $A \rightarrow B\ Q$ is { b, q}
- Those tokens that can follow A in some derivation from B Q is {c}

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

The set of tokens that predicts rule p:

{b, q, c}

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
$\quad \mid\ \lambda$		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

back

Computing predict(p)

predict(p: $X \to X_1X_2\ldots X_m$) // returns a set of tokens Slide 21

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

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

ans = *ans* ∪ **follow**(X); slide 22

return *ans*;

}

back

- **first**($X_1X_2\ldots X_m$) returns a set of tokens each of which is the **first token in a sentence** derived from $X_1X_2\ldots X_m$. Formally:

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 ($X_1X_2 \ldots X_m$)

```
first (X1X2...Xm) // returns a set of tokens
{   for each nonterminal X in the language
        visitedFirst[X] = false;
    ans = internalFirst(X1X2...Xm);
    return ans;
}
```

Computing first $(X_1 X_2 \ldots X_m)$

The main ideas for computing **internalFirst**$(X_1 X_2 \ldots X_{m:})$:

1. If $X_1 X_2 \ldots X_m = \lambda$, there is no first token. Return empty set.

   internalFirst(λ) returns $\varnothing$

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

   internalFirst(b B) returns b

3. If X_1 is a nonterminal

   i. Look at every rule for X_1 and find the first tokens of X_1.

   What does internalFirst(A C) do?

   Slide 25

   ii. If X_1 may derive empty, find the first tokens for $X_2 \ldots X_m$.

internalFirst$(X_1X_2\ldots X_m)$ //returns a set of tokens

```
{   if (m == 0)  return ∅;                              /* 1 */
    if (X1 is a terminal symbol)  return {X1}           /* 2 */
    /* X1 is a nonterminal */
    ans = ∅;
    if not visitedFirst[X1]
        visitedFirst[X1] = true;
        for the RHS of each rule for X1                 /* 3.i */
            ans = ans ∪ internalFirst(RHS);
    if symbolDerivesEmpty[X1]                           /* 3.ii */
        ans = ans ∪ internalFirst(X2…Xm);
    return ans;        }
```

$S \rightarrow A\ C$
$C \rightarrow c$
$\quad | \ \lambda$
$A \rightarrow a\ B\ C\ d$
$\quad | \ B\ Q$
$B \rightarrow b\ B$
$\quad | \ \lambda$
$Q \rightarrow q$
$\quad | \ \lambda$

Slide 25

internalFirst(B Q) returns {b, q}

Example first(B)

A	$\rightarrow$	B
	$\mid$	a
B	$\rightarrow$	A
	$\mid$	b

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

{ for each nonterminal X in the language

visitedFirst[X] = false;

ans = **internalFirst**$(X_1X_2\ldots X_m)$;

return *ans*;

}

visitedFirst

A	B
F	F

internalFirst(B) ➡ $\{a,b\}$

internalFirst(B)

/* X_1 = B i.e. a nonterminal */

ans = $\varnothing$;

if not visitedFirst[X_1]

visitedFirst[X_1] = true;

for the *RHS* of each rule for X_1

ans = *ans* $\cup$ **internalFirst**(*RHS*);

visitedFirst

A	B
F	T

internalFirst(A) $\cup$

internalFirst(b)

$$
\begin{array}{lll}
A & \rightarrow & B \\
  & | & a \\
B & \rightarrow & A \\
  & | & b
\end{array}
$$

internalFirst(A)

/* X_I = A i.e. a nonterminal */
ans = $\varnothing$;
if not visitedFirst[X_I]
    visitedFirst[X_I] = true;
    for the *RHS* of each rule for X_I
        ans = *ans* $\cup$ **internalFirst**(*RHS*);
if symbolDerivesEmpty[X_I]
    ans = *ans* $\cup$ **internalFirst**($X_2 \ldots X_m$);

return *ans*;

internalFirst(B) $\cup$
internalFirst(a)

visitedFirst:

A	B
T	T

symbolsDerivesEmpty:

A	B
F	F

internalFirst(A) $\Rightarrow$ **{*a*}**

$$
\begin{aligned}
A &\rightarrow B \\
&\mid a \\
B &\rightarrow A \\
&\mid b
\end{aligned}
$$

internalFirst(B)

```
/* X_l = B i.e. a nonterminal */
ans = ∅;
if not visitedFirst[X_l]
      visitedFirst[X_l] = true;
      for the RHS of each rule for X_l
            ans = ans ∪ internalFirst(RHS);
if symbolDerivesEmpty[X_l]
      ans = ans ∪ internalFirst(X_2…X_m);

return ans;   ➡   ∅
```

visitedFirst:

A	B
T	T

symbolsDerivesEmpty:

A	B
F	F

internalFirst(B) ➡ $\varnothing$

Computing follow(X)

Slide 26

- **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*;

}

back

Main ideas of internalFollow(X)

How do we find what tokens may follow X?

1. Find each occurrence of X in all RHS, E.g. what may follow B:

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

   $A \rightarrow B\ Q$

   $B \rightarrow b\ \ B$

2. For each such occurrence, find the **first tokens** of the string after X. If this string derives empty, call internalFollow(LHS) to find what tokens follow the LHS nonterminal. E.g. in $A \rightarrow B\ Q$.

$S \rightarrow A\ C$

$C \rightarrow c$

$\quad |\ \ \lambda$

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

$\quad |\ \ B\ Q$

$B \rightarrow b\ B$

$\quad |\ \ \lambda$

$Q \rightarrow q$

$\quad |\ \ \lambda$

back

Computing follow(X)

E.g. internalFollow(B)

Occurrence of B in 'A → a B C d'
tail = 'C d' first(tail) returns {c, d} **ans = ∅ ∪ {c, d}**

internalFollow(*Y*) // *Y* is a nonterminal

```
{   ans = ∅;
    if not visitedFollow[Y]
        visitedFollow[Y] = true;
        for each occurrence of Y in the RHS of all rules
            tail = stream of symbols that appear after Y ;
            ans = ans ∪ first(tail);
            if allDeriveEmpty(tail)
                target = LHS of the rule;
                ans = ans ∪ internalFollow(target);
    return ans;
}
```

Occurrence of B in 'A → B Q'
tail = 'Q' first(Q) returns {q} **ans = {c, d} ∪ {q}** target = 'A' internalFollow(A) returns {c} **ans = {c, d, q} ∪ {c}**

Occurrence of B in 'B → b B'
… **ans = {c, d, q} ∪ ∅**

Computing follow(A)

allDeriveEmpty(β) // β is a stream of symbols

```
{
   for each symbol X in β
      if X is a terminal or not symbolDerivesEmpty[X]
         return false;
   return true;
}
```

back

E.g. allDerivesEmpty(tail) is called when tail = "C d". It will return false.

	S	C	A	B	Q
symbolDerivesEmpty	T	T	T	T	T

Obtaining LL(1) Grammars

- LL(1) requires a unique combination of a nonterminal and a lookahead symbol to decide which rule to use. slide21
- Two common categories of production 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 $\rightarrow$ number plus *Expr*
$\quad\quad$ | number

Obtaining LL(1) Grammars

Example
$\mathit{Expr} \rightarrow$ number plus Expr $\vert$ Factor
$\mathit{Factor} \rightarrow$ number

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

Example
$\mathit{Expr} \rightarrow$ number $\mathit{Expr'}$
$\mathit{Expr'} \rightarrow$ plus Expr $\vert$ 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	→	*StmtList* semicolon *Stmt*
	|	*Stmt*
…		

The method that parses *StmtList*, parseStmtList() will call itself repeatedly 'forever'.

Slide 21

Obtaining LL(1) Grammars

1. Change left recursion to right recursion:

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

2. Remove the common prefix:

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

3. May want to remove mutual recursion:

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

Slide 17

Slide 21

Syntactic Error Recovery

- A compiler should produce a useful set of diagnostic messages when presented with a faulty program.
- Thus after an error is detected it is desirable to recover from it and continue the syntax analysis.
- Semantic analysis and code generation will be disabled.
- 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

d should follow B

```
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} ∪ termset);  …
   else
         error("expected an a or b");
         skip input till a symbol in termset is found
}
```

q should follow B

$A \to$ **a B d e**

$\quad |$ **B Q e**

$B \to b$

$Q \to q$

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

Bottom-up Parsing

- Recall: Slide 16

- 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
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 $

back

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.
- 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.

Slide 23, chap 2

Slide 47

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 shown in Slide 45 ..

back

Slide 47

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 shown in Slide 45

back

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 shown in Slide 45

Slide 47

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						

back

Figure 6.5: Parse table for the grammar shown in Slide 45

Slide 47

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						

back

Figure 6.5: Parse table for the grammar shown in Slide 45

Slide 47

shift state 3

Reduce by rule 8

input: $

0	4 S	8 $

1. Start → S $

input: Start $

0

accept

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 shown in Slide 45

Slide 47

Classes of Bottom-up Parsers

- In practice, bottom-up parsers only depend on a single token of lookahead.
- They can also be described as finite automata (though of a more complicated kind than DFAs)
- There are several methods for constructing parse tables:
  - 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 analyzes 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.

Slide 52

- During parsing, we want to keep track of where we are in the grammar.
- To do this, we use **LR(0) items**: an LR(0) item is **a grammar rule with a marker "•"** showing the current progress of the parser in recognizing the RHS of the rule.

LR(0) Automata and Table Construction

- Symbols before the • have already been seen. The first symbol after • is what we expect next. For examples,
  E $\rightarrow$ • plus E E, E $\rightarrow$ plus • E E, E $\rightarrow$ plus E • E
- For an item $A \rightarrow \alpha \bullet \beta$, the item is called ***initial*** if $\alpha \rightarrow \lambda$, and ***final*** if $\beta \rightarrow \lambda$. A final item for the start symbol is called accepting.

E. g.

Start $\rightarrow$ • S \$ // an initial item

S $\rightarrow$ A C • // a final item

LR(0) Automata and Table Construction

- An 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 following the marker, we add the initial items for A, i.e., items for all rules of A with the marker at the beginning of the RHS. This is called taking ***the closure of the item***.

E.g., For an item $S \to \bullet A$,

we add

$A \to \bullet a\ B\ d$, $A \to \bullet B\ Q$,

$B \to \bullet b$

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

$$
\begin{aligned}
S &\to A \\
A &\to a\ B\ d \\
&\quad \mid B\ Q \\
B &\to b \\
Q &\to q
\end{aligned}
$$

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:

$$\text{Start} \rightarrow \bullet\ \text{E}\ \ \$$$

$$\begin{aligned} \text{Start} &\rightarrow \text{E}\ \ \$ \\ \text{E} &\rightarrow \text{plus}\ \ \text{E}\ \ \text{E} \\ &\quad\mid\ \ \text{num} \end{aligned}$$

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

0

$$\boxed{\begin{aligned} &\text{Start} \rightarrow \bullet\ \text{E}\ \ \$ \\ &\text{E} \rightarrow \bullet\ \text{plus}\ \ \text{E}\ \ \text{E} \\ &\text{E} \rightarrow \bullet\ \text{num} \end{aligned}}$$

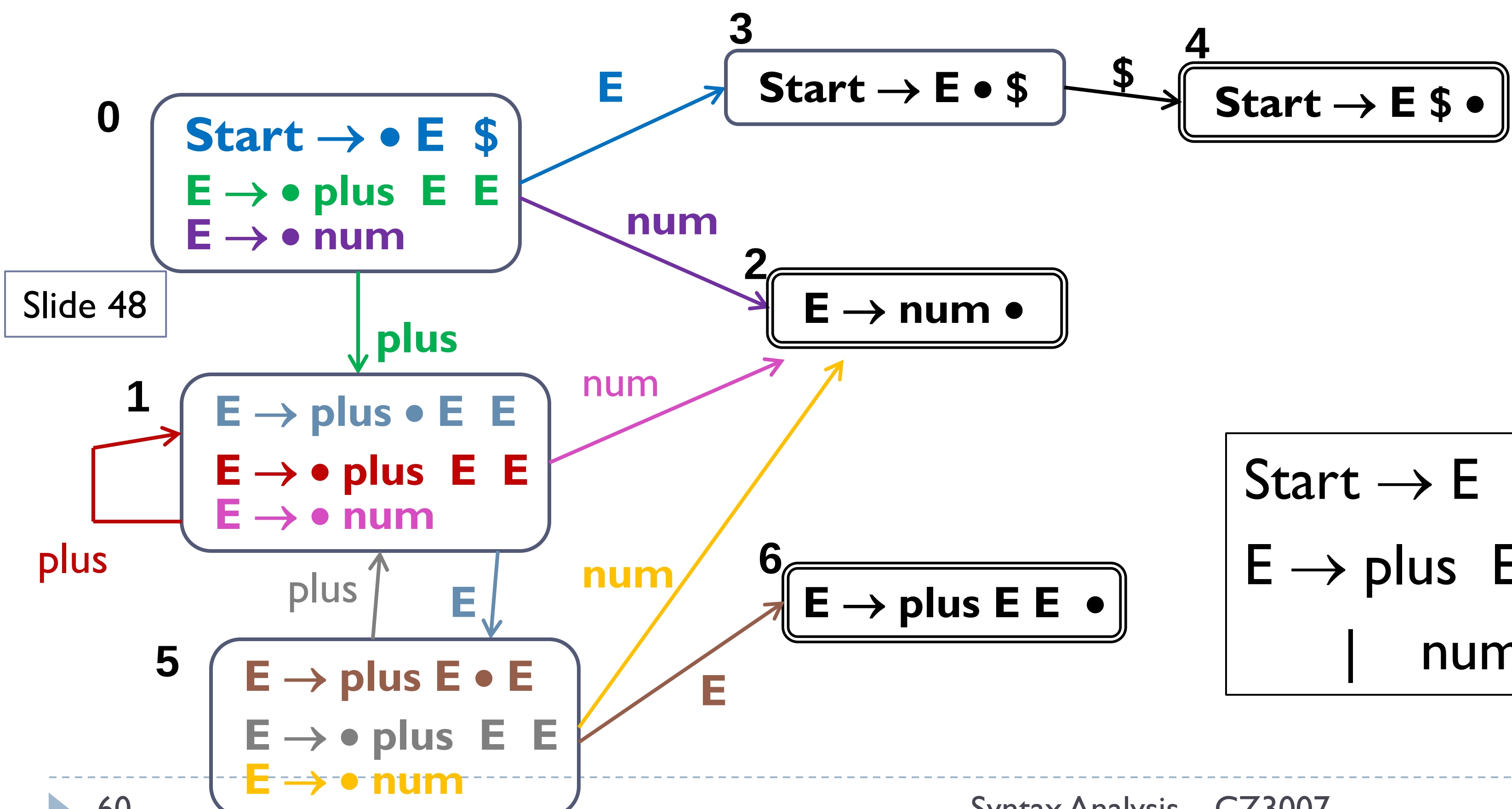
In slides 48, 51, 52, state 0 is on TOS

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 an item of the form $A \to \alpha \bullet \gamma\ \beta$ with $A \to \alpha\ \gamma \bullet \beta$
  - throw away all other items;
  - take closure
- Transitions are labelled by γ which is either a terminal or a nonterminal.
- If there is a final item of the form $A \to \alpha \bullet$, we can **reduce**.
- An accepting state is one that contains a final item for start symbol.

LR(0) Automata and Table Construction

- Continue from State 0 of our finite automata in slide 58, 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

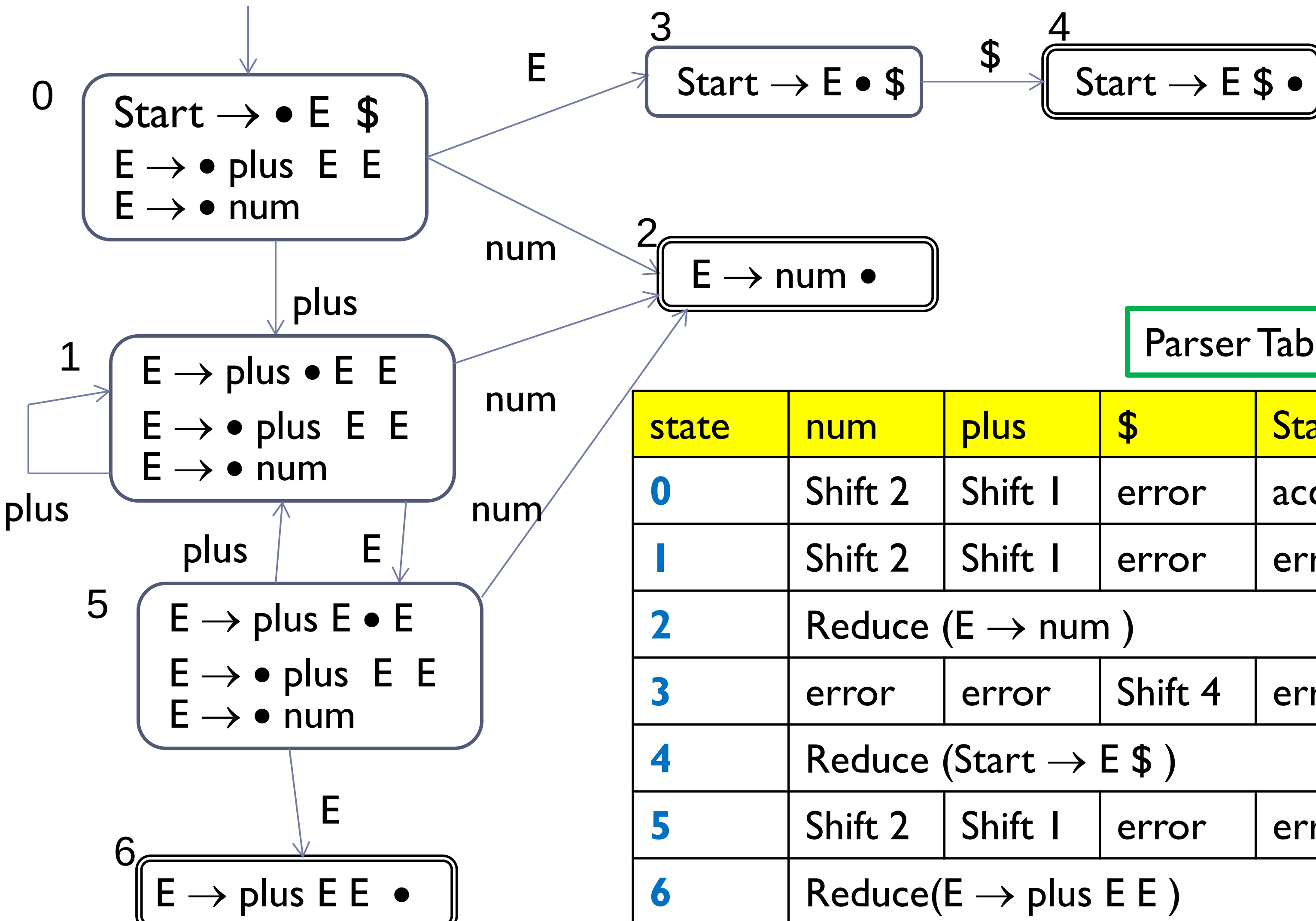
- The LR(0) automaton needs to be used together with a stack of states; this kind of automaton is known as a ***pushdown automaton.***
- The state of the parser is the state on top of the parser stack. Slide 51
- Actions taken during bottom-up parsing are of four types:
  - **shift**(i): consume next input symbol, push state i onto state stack; Slide 47
  - **reduce**($A \rightarrow \gamma$): reduce by rule $A \rightarrow \gamma$, 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 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 all cells of row s
  - For cell (0, StartSymbol), enter **accept**
  - Enter **error** into any remaining empty cells

back

Example



back

Parser Table

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

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 $A \rightarrow \beta \bullet \gamma$ (a **shift()** action) and a final item $A \rightarrow \beta \bullet$ (a **reduce()** action)
  - E.g. A state has these two items:
    - IfStatement $\rightarrow$ IF Cond THEN StatList $\bullet$ ELSE StatList
    - IfStatement $\rightarrow$ IF Cond THEN StatList $\bullet$
- Example:

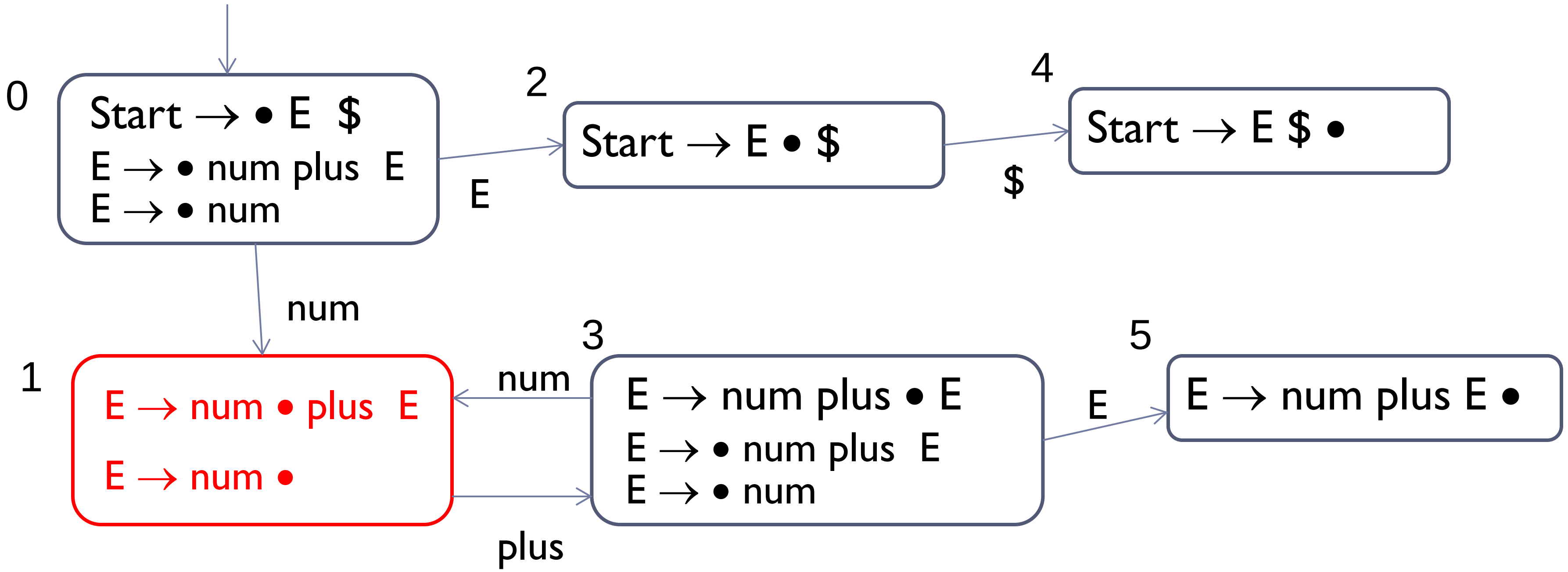
  Start $\rightarrow$ E $

  E $\rightarrow$ num plus E

  | num

back

Example: shift-reduce conflict



state	num	plus	$	Start	E
0	Shift 1	error	error	accept	Shift 2
1		Shift/reduce?			
2	…				

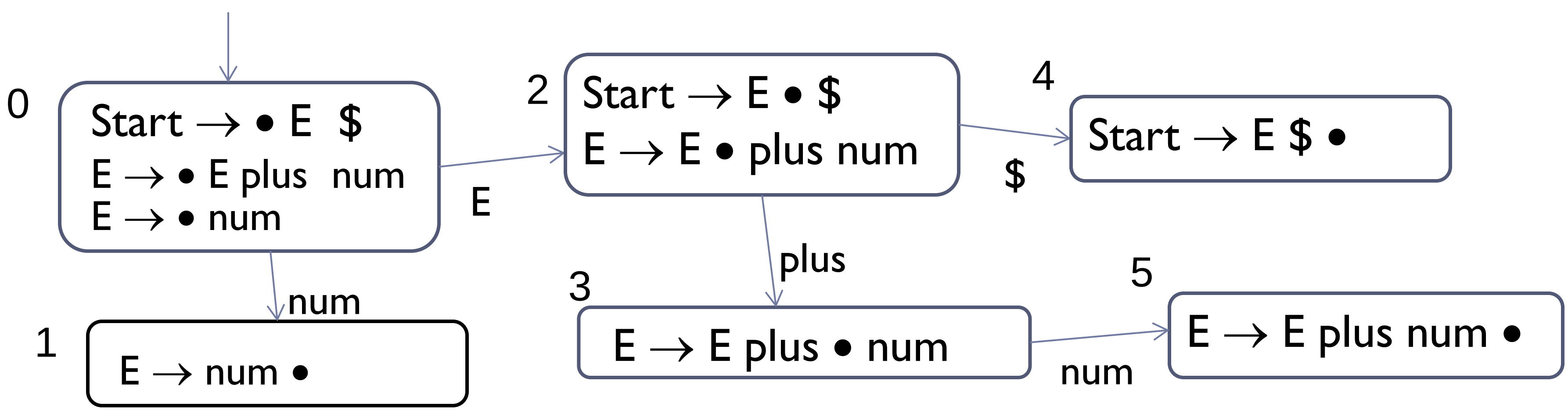
back

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,

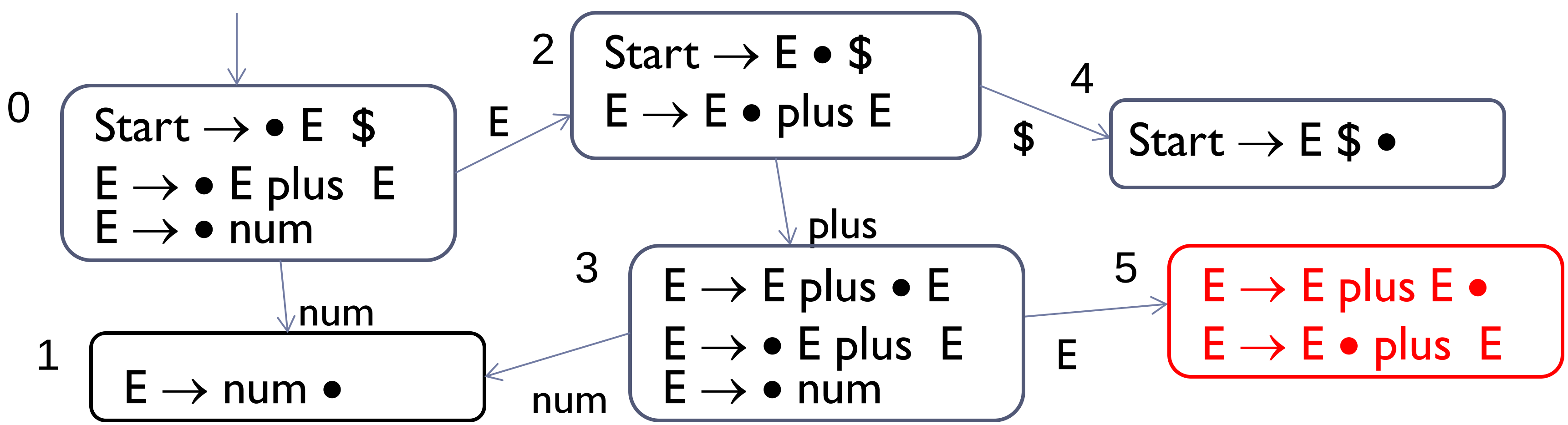
  Start → E $

  E → E plus E

  | num

Rewriting the grammar will solve the problem in this case.

LR(0) automaton:



LALR(1) – Look Ahead LR with one token lookahead

back

- 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

Compare: slide 62

- **itemFollow()** keeps track of the symbols that can follow the ***item*** after reduction *from this state*.

Slide 66

LALR Propagation Graph

- Consider the LR(0) table, the pair (s, $A \rightarrow \alpha \bullet \beta$) suffices to identify an item $A \rightarrow \alpha \bullet \beta$ that occurs in state *s*.

Slide 63

- Each item in an LR(0) table is represented by a vertex in the **LALR propagation graph**.
- We will compute itemFollow() for each 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. Initialize all itemFollow[*v*] to ∅.
   4. Initialize itemFollow[(0, StartSymbol Productions)] = {$}

B. Build the propagation graph

C. Propagate itemFollow[ ]

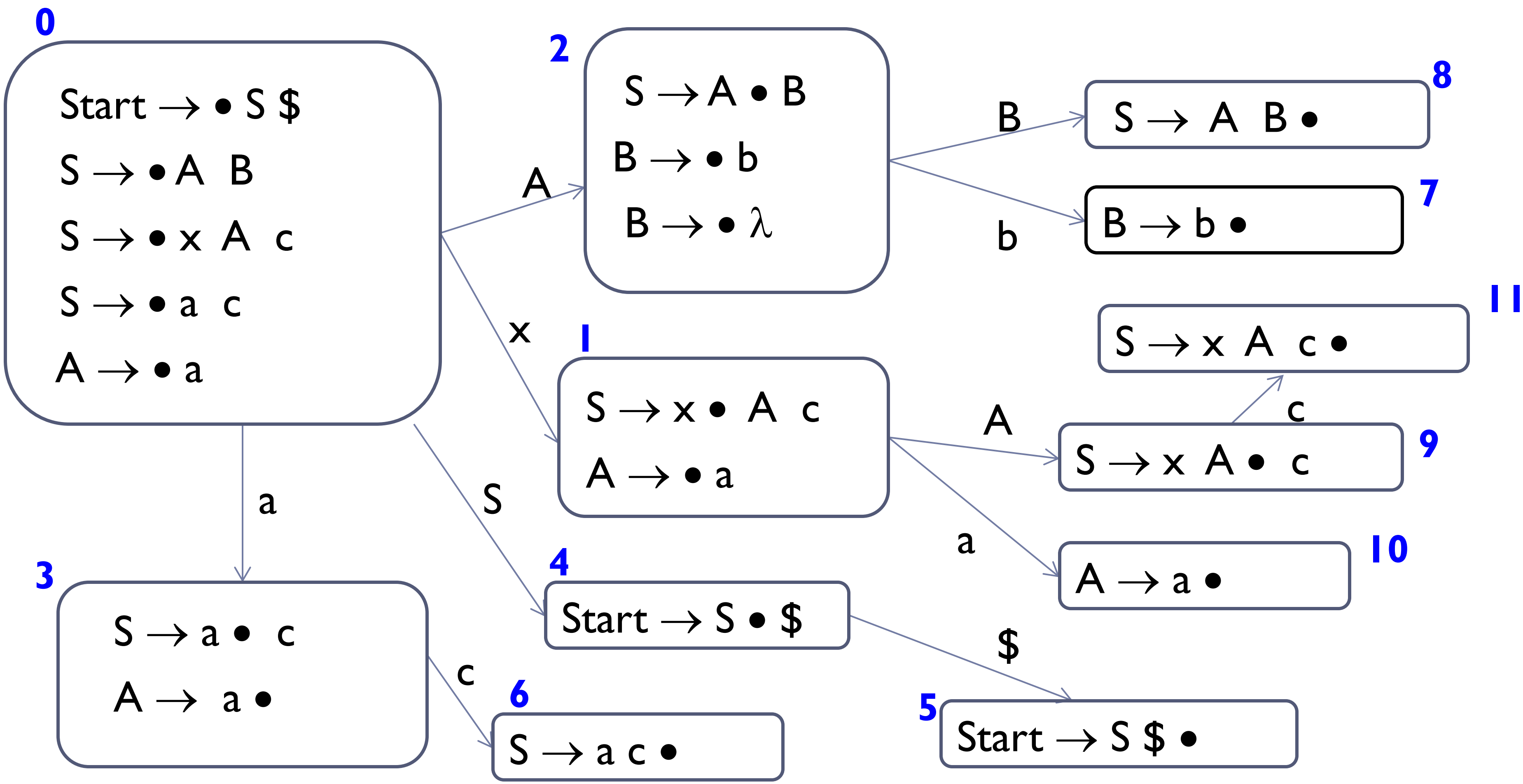
back

Generating the Propagation Graph (Example)

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

back

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



A: Setup (create vertices, initialize 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	$\emptyset$
11	3, S $\rightarrow$ a $\bullet$ c	$\emptyset$
12	3, A $\rightarrow$ a $\bullet$	$\emptyset$
8	2, S $\rightarrow$ A $\bullet$ B	$\emptyset$
9	2, B $\rightarrow \bullet$ b	$\emptyset$
10	2, B $\rightarrow \bullet\ \lambda$	$\emptyset$
6	1, S $\rightarrow$ x $\bullet$ A c	$\emptyset$
7	1, A $\rightarrow \bullet$ a	$\emptyset$
13	4, Start $\rightarrow$ S $\bullet$ \$	$\emptyset$
15	6, S $\rightarrow$ a c $\bullet$	$\emptyset$
17	8, S $\rightarrow$ A B $\bullet$	$\emptyset$
16	7, B $\rightarrow$ b $\bullet$	$\emptyset$
20	11, S $\rightarrow$ x A c $\bullet$	$\emptyset$
18	9, S $\rightarrow$ x A $\bullet$ c	$\emptyset$
19	10, A $\rightarrow$ a $\bullet$	$\emptyset$
14	5, Start $\rightarrow$ S \$ $\bullet$	$\emptyset$

B:Building the Propagation Graph (ideas)

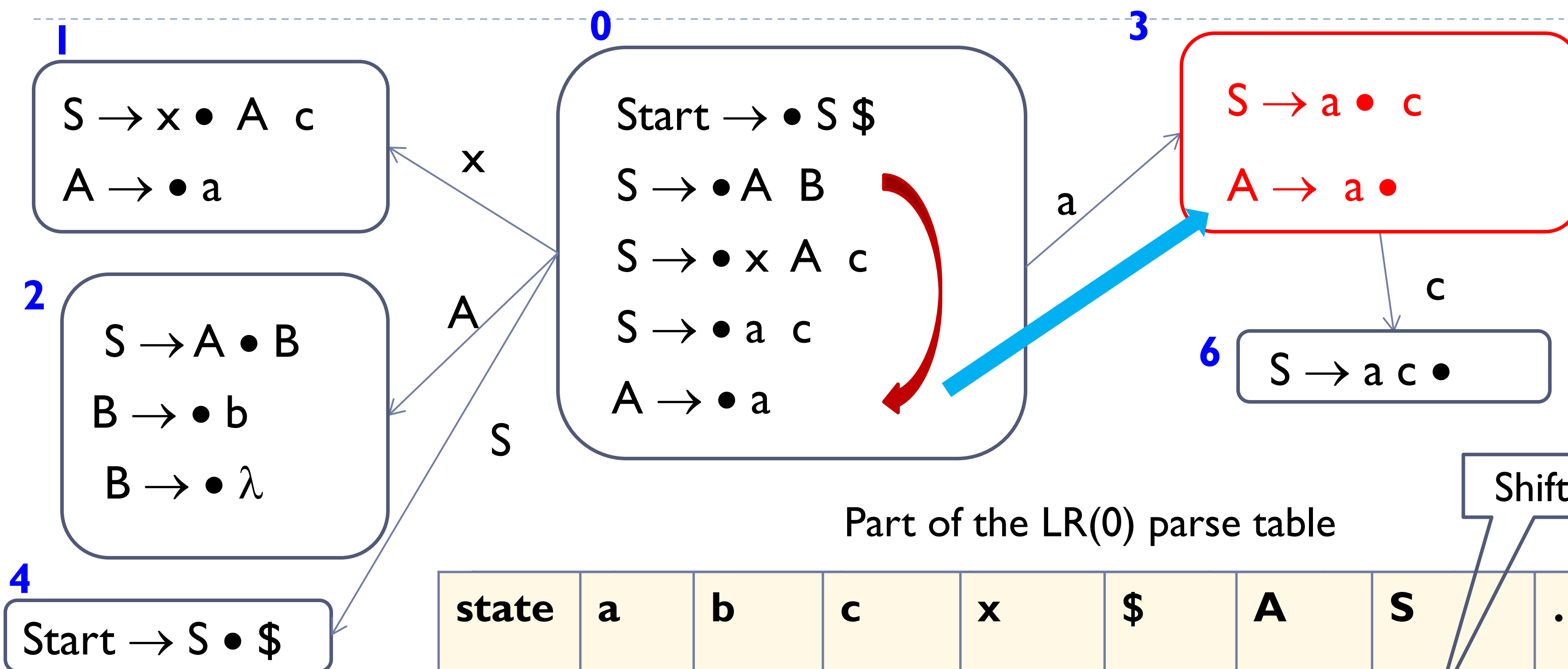
- For item $(s, A \rightarrow \alpha \bullet B\gamma)$, any symbol in First($\gamma$) can follow each closure item $(s, B \rightarrow \bullet\delta)$ (2)
  - Put First(γ) into the itemFollow set of each initial item of B
- A propagation edge is placed from an item $(s, A \rightarrow \alpha \bullet B\gamma)$ to an item $(t, A \rightarrow \alpha B \bullet \gamma)$ (1)
- For item $(s, A \rightarrow \alpha \bullet B\gamma)$, when $\gamma \Rightarrow^* \lambda$, any symbol that can follow A can also follow B (3)
  - Place a propagation edge from $(s, A \rightarrow \alpha \bullet B\gamma)$ to $(s, B \rightarrow \bullet\delta)$

Slide 66

back

itemFollow() keeps track of the symbols that can follow the ***item*** after reduction *from this state*

itemFollow() keeps track of the symbols that can follow the ***item*** after reduction *from this state*



Part of the LR(0) parse table

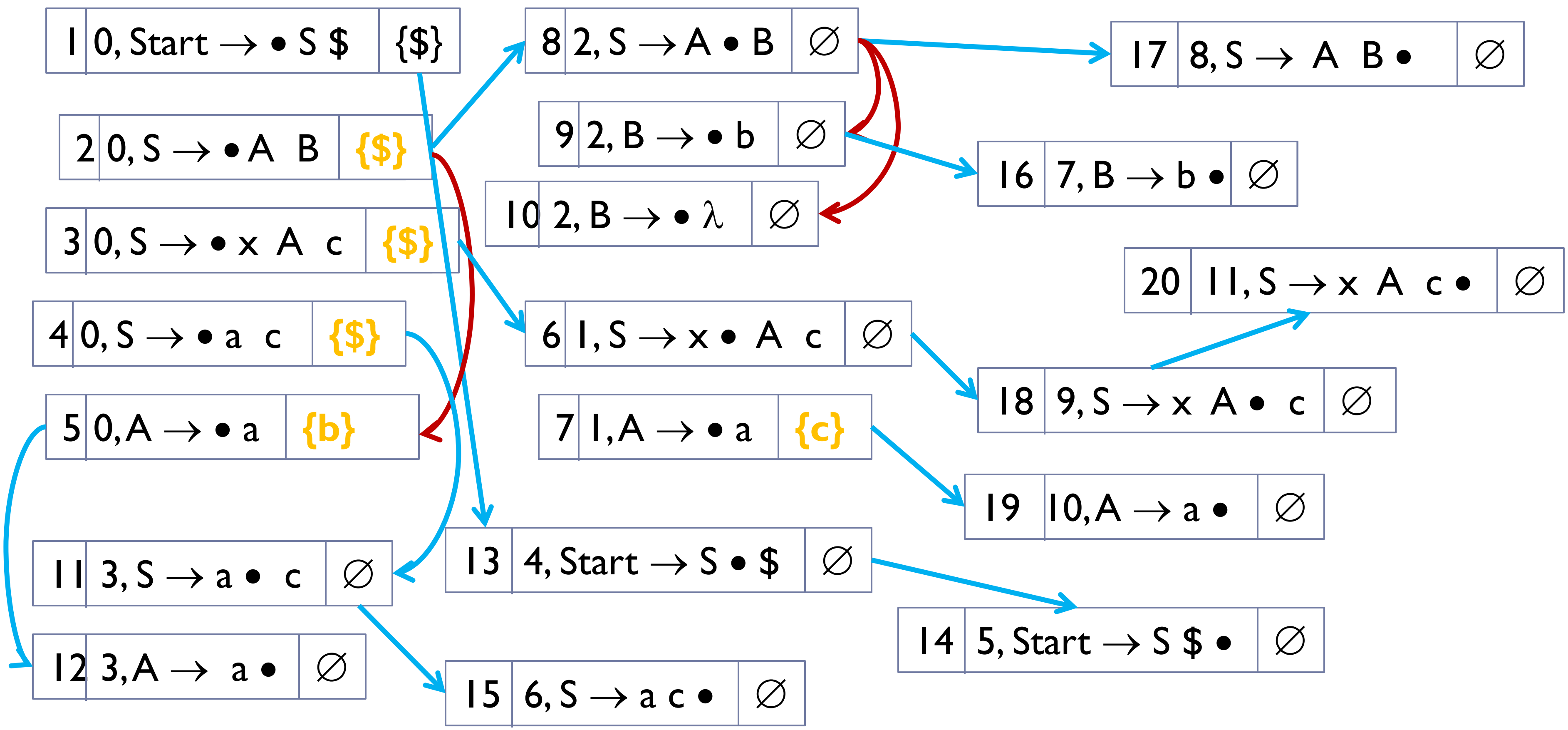
Shift 4

state	a	b	c	x	$	A	S	…
0	S3	err	err	S1	err	S2	S4	…
1								
2								
3	err	R5	S6	err	R5	err	err	

Input scenarios:
1) a $
2) a b $
3) a c $

Slide 72

B: Building the Propagation Graph



Slide 75

B: Building the Propagation Graph

For each state *s*

For each item of form $A \rightarrow \alpha \bullet B\ \gamma$ in *s*

vertex $u = (s,\ A \rightarrow \alpha \bullet B\ \gamma)$

s' = Table[s][B]

vertex $v = (s', A \rightarrow \alpha B \bullet \gamma)$, successor of *u* on shift(B)

Add edge (*u*, *v*) 1

For each vertex *w* of form $(s, B \rightarrow \bullet\ \delta)$

Add **first**(γ) to itemFollow[*w*] 2

if **allDeriveEmpty**(γ)

add edge (*u*, *w*) 3

Slide 36

C: Propagating itemFollw [ ]

Slide 71

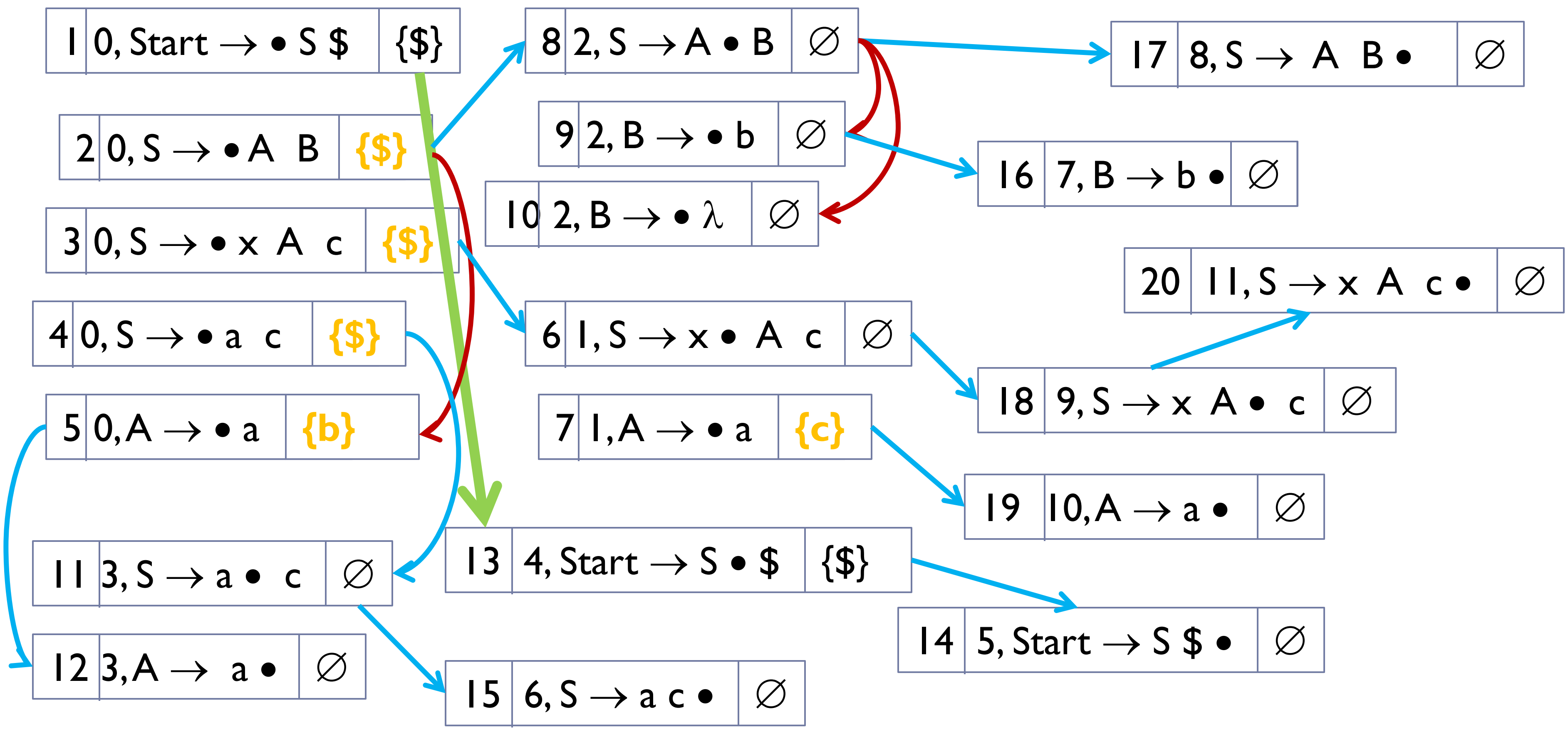
While (making progress)

For each edge (u,v)

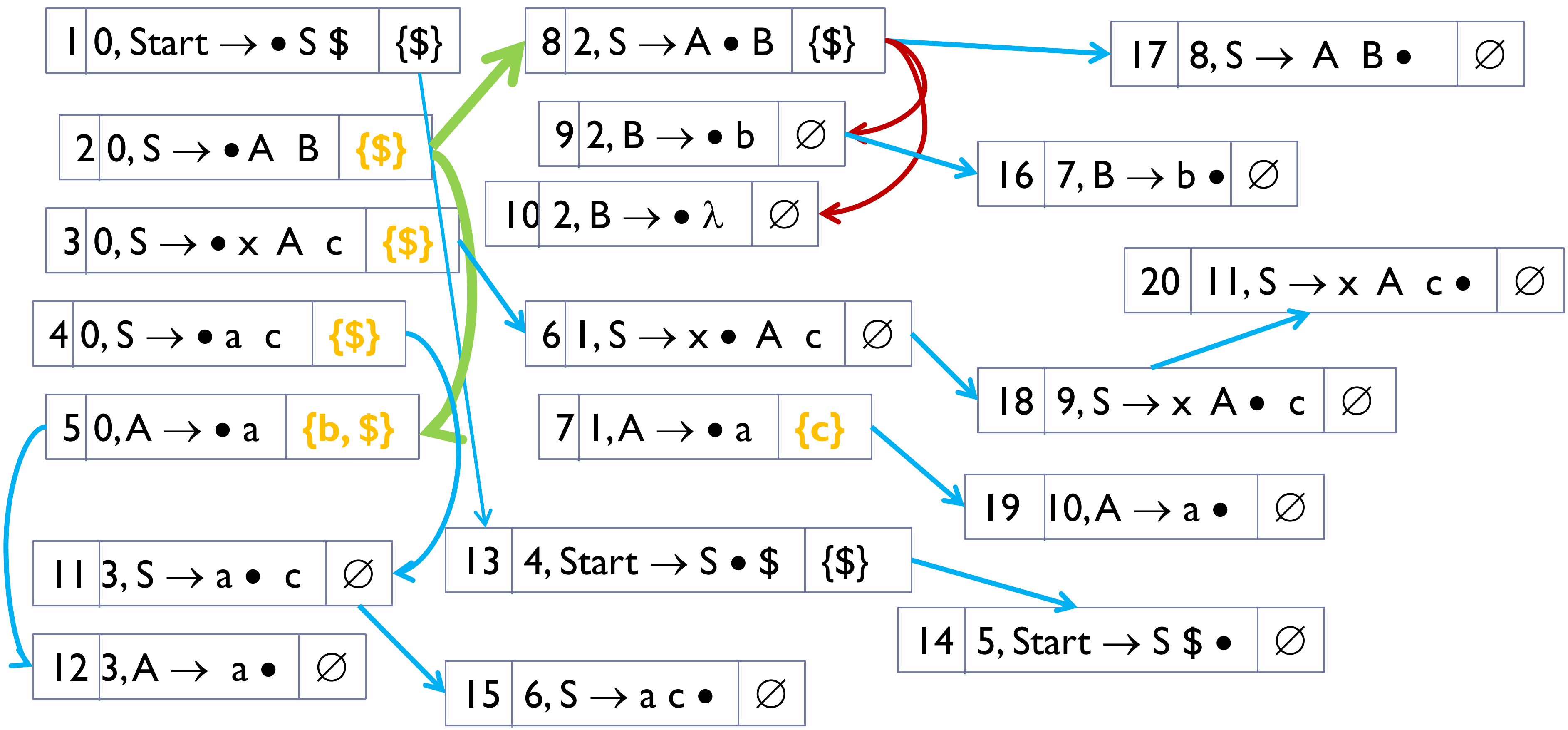
add ItemFollow(u) to ItemFollow(v) 4

- In general, multiple passes can be required
- In practice, LALR(1) lookahead computations converge quickly, usually in one or two passes
- LALR(1) is a powerful parsing method
- LALR(1) grammars are available for all popular programming languages

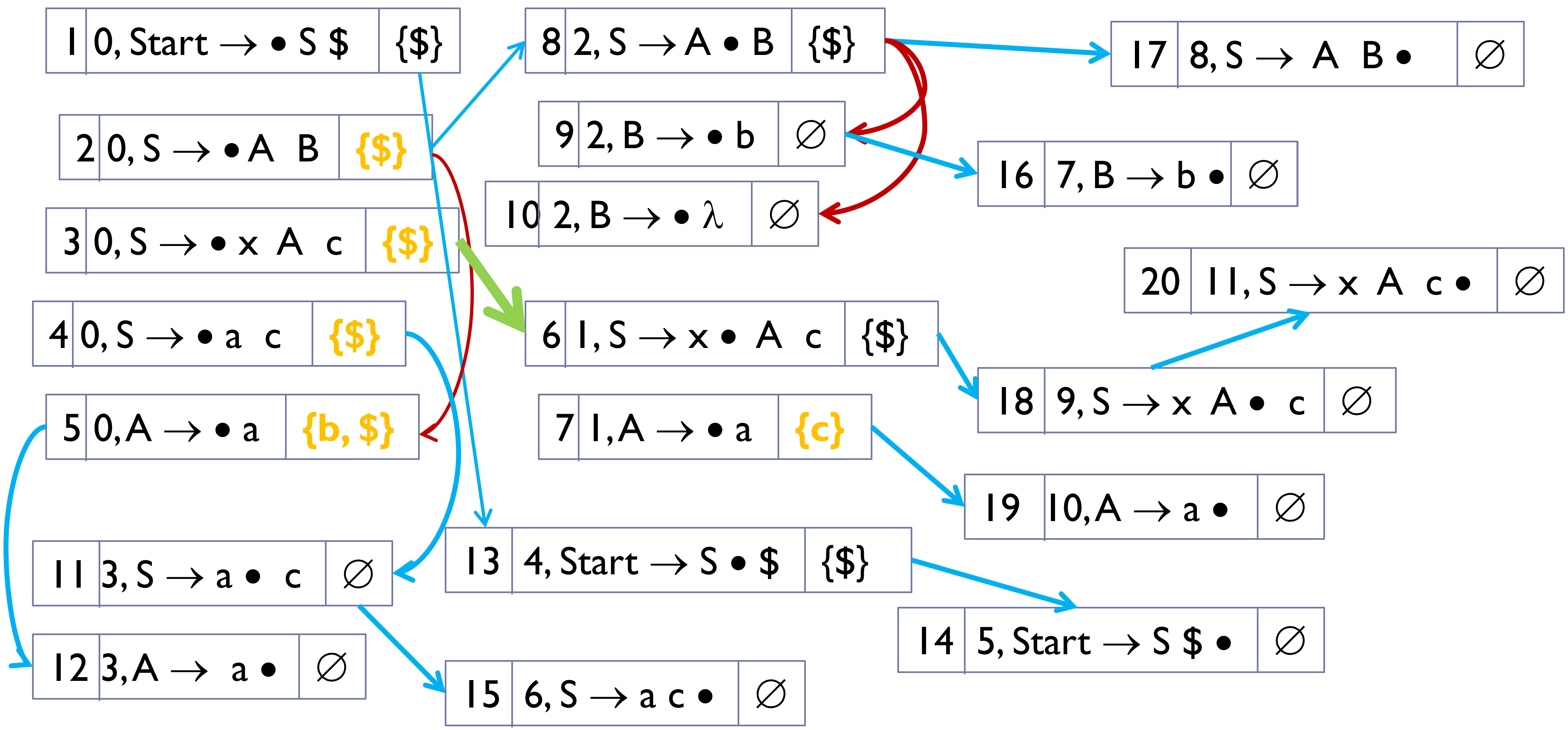
C: Propagating itemFollw [ ]



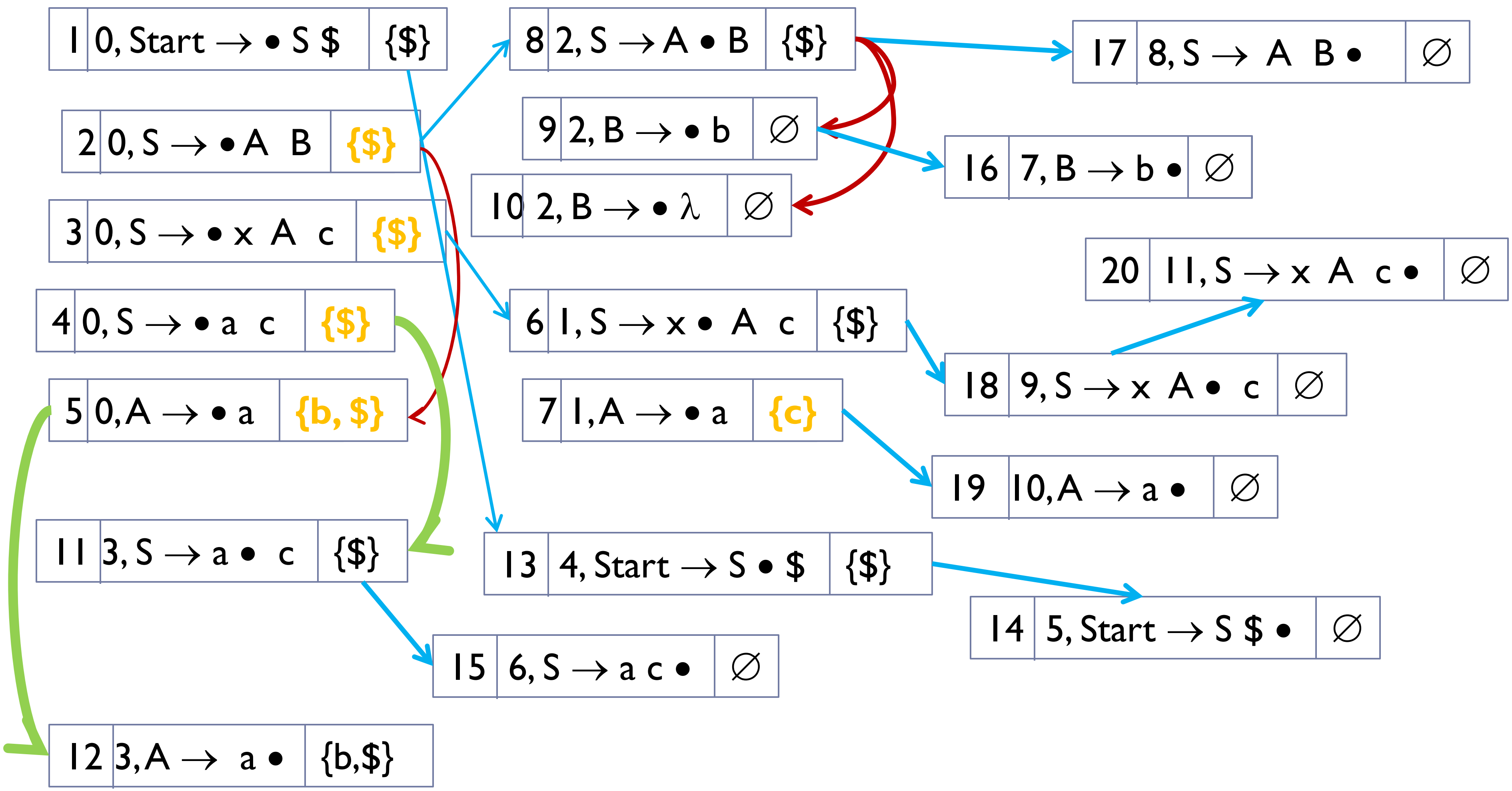
C: Propagating itemFollw [ ]



C: Propagating itemFollw [ ]



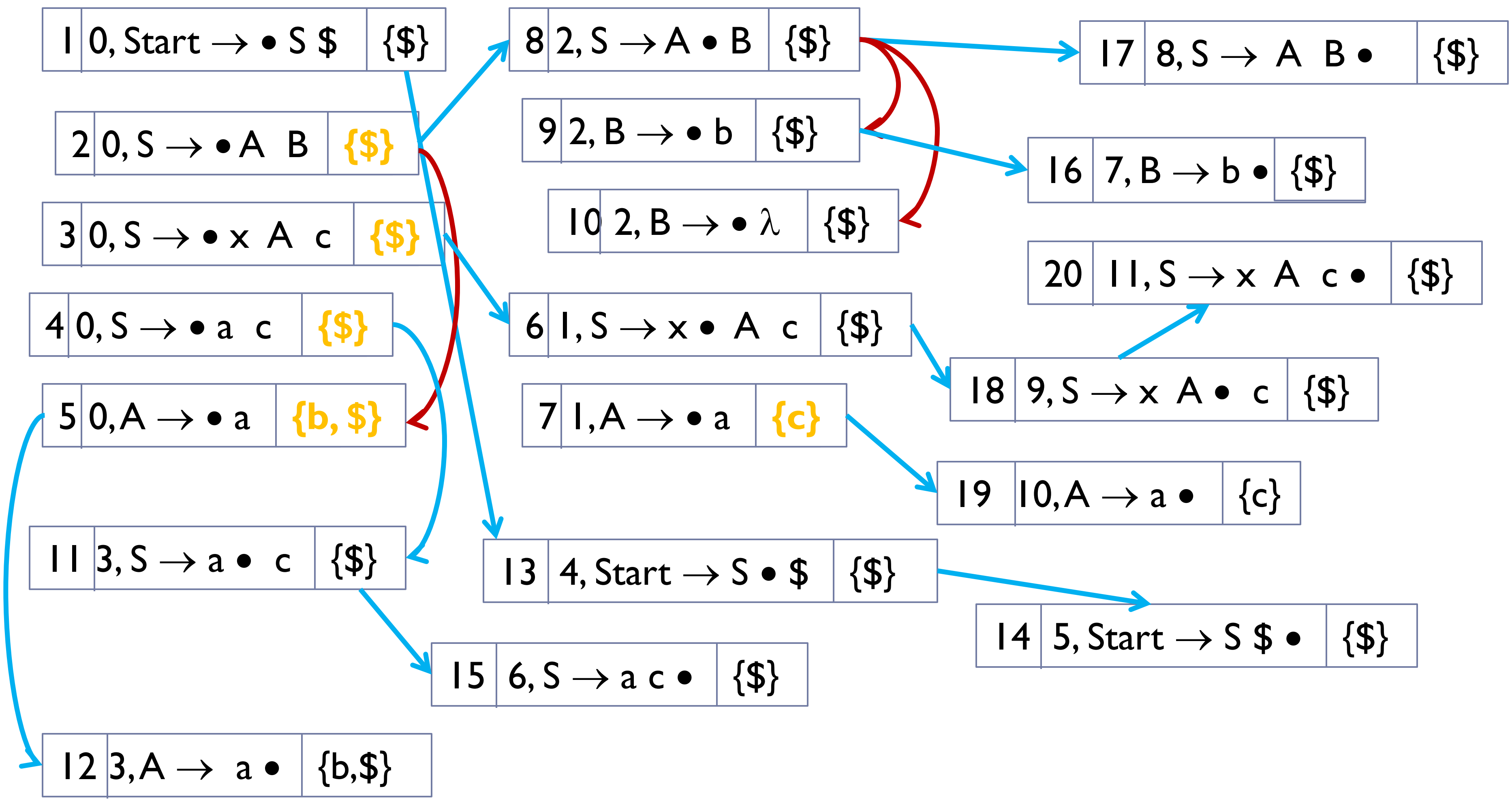
C: Propagating itemFollw [ ]



C: Propagating itemFollw [ ]

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

C: Propagating itemFollw [ ]



The Result:

The LALR(1) parse table:

Start $\rightarrow$ S \$

S $\rightarrow$ A B

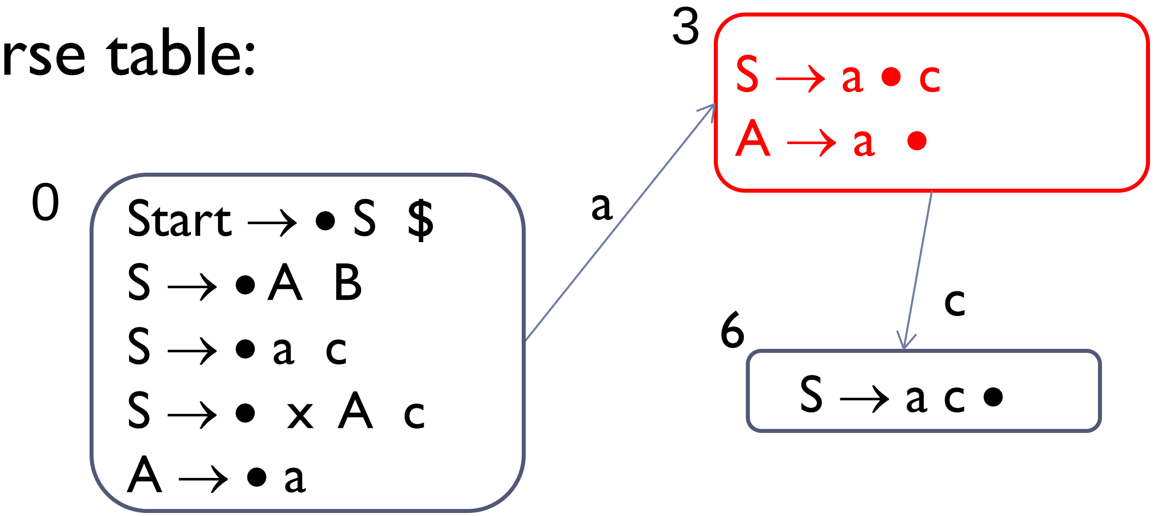
| a c

| x A c

A $\rightarrow$ a

B $\rightarrow$ b

| λ



Slide 69

itemFollow(3, A $\rightarrow$ a •) = {b, \$}

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

back

Parser Generator Beaver

- Beaver is a LALR(1) parser generator that generates parsers written in Java.
- 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 $\rightarrow$.

Slide 6

- 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.
- 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
     ;
```

Some review questions/tasks

1. What does a syntax analyzer do?
2. What are the input and output of a syntax analyzer?
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 6 (reference slide 21). Assume the methods peek(), match() and predict() are provided.
5. Follow the bottom-up parsing engine on slide 47 and use the parse table on slide 48 for the grammar on slide 45 to trace the parsing process for the input "adc$".