Exercise 4 4 10 Show how having lled in the table as in Exercise 4 4 9 we can in O n time recover a parse tree for a_1a_2 a_n *Hint* modify the table so it records for each nonterminal A in each table entry T_{ij} some pair of nonterminals in other table entries that justi ed putting A in T_{ij}

Exercise 4 4 11 Modify your algorithm of Exercise 4 4 9 so that it will nd for any string the smallest number of insert delete and mutate errors each error a single character needed to turn the string into a string in the language of the underlying grammar

stmt		**if** *e* **then** *stmt stmtTail*
	$\mid$	**while** *e* **do** *stmt*
	$\mid$	**begin** *list* **end**
	$\mid$	*s*
stmtTail		**else** *stmt*
	$\mid$	
list		*stmt listTail*
listTail		*list*
	$\mid$	

Figure 4 24 A grammar for certain kinds of statements

Exercise 4 4 12 In Fig 4 24 is a grammar for certain statements You may take e and s to be terminals standing for conditional expressions and other statements respectively If we resolve the con ict regarding expansion of the optional else nonterminal *stmtTail* by preferring to consume an **else** from the input whenever we see one we can build a predictive parser for this grammar Using the idea of synchronizing symbols described in Section 4 4 5

a Build an error correcting predictive parsing table for the grammar

b Show the behavior of your parser on the following inputs

i **if** *e* **then** *s* **if** *e* **then** *s* **end**
ii **while** *e* **do begin** *s* **if** *e* **then** *s* **end**

4 5 Bottom Up Parsing

A bottom up parse corresponds to the construction of a parse tree for an input string beginning at the leaves the bottom and working up towards the root the top It is convenient to describe parsing as the process of building parse trees although a front end may in fact carry out a translation directly without building an explicit tree The sequence of tree snapshots in Fig 4 25 illustrates

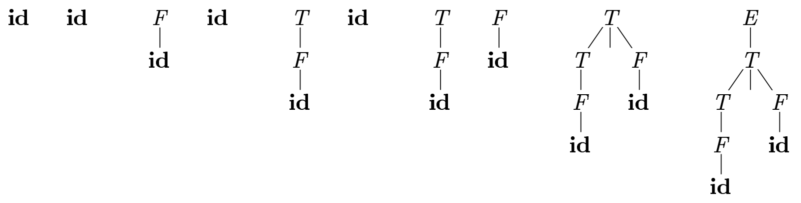

Figure 4 25 A bottom up parse for **id** **id**

a bottom up parse of the token stream **id** **id** with respect to the expression grammar 4 1

This section introduces a general style of bottom up parsing known as shift reduce parsing The largest class of grammars for which shift reduce parsers can be built the LR grammars will be discussed in Sections 4 6 and 4 7 Although it is too much work to build an LR parser by hand tools called automatic parser generators make it easy to construct e cient LR parsers from suitable gram mars The concepts in this section are helpful for writing suitable grammars to make e ective use of an LR parser generator Algorithms for implementing parser generators appear in Section 4 7

4 5 1 Reductions

We can think of bottom up parsing as the process of reducing a string w to the start symbol of the grammar At each *reduction* step a speci c substring matching the body of a production is replaced by the nonterminal at the head of that production

The key decisions during bottom up parsing are about when to reduce and about what production to apply as the parse proceeds

Example 4 37 The snapshots in Fig 4 25 illustrate a sequence of reductions the grammar is the expression grammar 4 1 The reductions will be discussed in terms of the sequence of strings

$$\mathbf{id}\quad \mathbf{id}\quad F\quad \mathbf{id}\quad T\quad \mathbf{id}\quad T\quad F\quad T\quad E$$

The strings in this sequence are formed from the roots of all the subtrees in the snapshots The sequence starts with the input string **id** **id** The rst reduction produces F **id** by reducing the leftmost **id** to F using the production F **id** The second reduction produces T **id** by reducing F to T

Now we have a choice between reducing the string T which is the body of E T and the string consisting of the second **id** which is the body of F **id** Rather than reduce T to E the second **id** is reduced to F resulting in the string T F This string then reduces to T The parse completes with the reduction of T to the start symbol E □

By de nition a reduction is the reverse of a step in a derivation recall that in a derivation a nonterminal in a sentential form is replaced by the body of one of its productions The goal of bottom up parsing is therefore to construct a derivation in reverse The following corresponds to the parse in Fig 4 25

$$E \quad T \quad T \ F \quad T \ \mathbf{id} \quad F \ \mathbf{id} \quad \mathbf{id} \ \mathbf{id}$$

This derivation is in fact a rightmost derivation

4 5 2 Handle Pruning

Bottom up parsing during a left to right scan of the input constructs a right most derivation in reverse Informally a handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation

For example adding subscripts to the tokens **id** for clarity the handles during the parse of $\mathbf{id}_1 \quad \mathbf{id}_2$ according to the expression grammar 4 1 are as in Fig 4 26 Although T is the body of the production $E \quad T$ the symbol T is not a handle in the sentential form $T \quad \mathbf{id}_2$ If T were indeed replaced by E we would get the string $E \quad \mathbf{id}_2$ which cannot be derived from the start symbol E Thus the leftmost substring that matches the body of some production need not be a handle

Right Sentential Form	Handle	Reducing Production
$\mathbf{id}_1 \quad \mathbf{id}_2$	$\mathbf{id}_1$	$F \quad \mathbf{id}$
$F \quad \mathbf{id}_2$	F	$T \quad F$
$T \quad \mathbf{id}_2$	$\mathbf{id}_2$	$F \quad \mathbf{id}$
$T \ F$	$T \ F$	$T \quad T \quad F$
T	T	$E \quad T$

Figure 4 26 Handles during a parse of $\mathbf{id}_1 \quad \mathbf{id}_2$

Formally if $S \ \underset{rm}{\ } \ Aw \ \underset{rm}{\ } \ w$ as in Fig 4 27 then production A in the position following is a *handle* of w Alternatively a handle of a right sentential form is a production A and a position of where the string may be found such that replacing at that position by A produces the previous right sentential form in a rightmost derivation of

Notice that the string w to the right of the handle must contain only terminal symbols For convenience we refer to the body rather than A as a handle Note we say a handle rather than the handle because the grammar could be ambiguous with more than one rightmost derivation of w If a grammar is unambiguous then every right sentential form of the grammar has exactly one handle

A rightmost derivation in reverse can be obtained by handle pruning That is we start with a string of terminals w to be parsed If w is a sentence

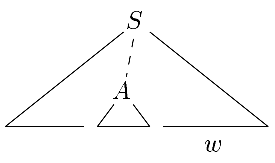

Figure 4 27 A handle A in the parse tree for w

of the grammar at hand then let w $_n$ where $_n$ is the nth right sentential form of some as yet unknown rightmost derivation

$$S \quad _0 \underset{rm}{} \quad _1 \underset{rm}{} \quad _2 \underset{rm}{} \quad \underset{rm}{} \quad _{n\ 1} \underset{rm}{} \quad _n \quad w$$

To reconstruct this derivation in reverse order we locate the handle $_n$ in $_n$ and replace $_n$ by the head of the relevant production A_n $_n$ to obtain the previous right sentential form $_{n\ 1}$ Note that we do not yet know how handles are to be found but we shall see methods of doing so shortly

We then repeat this process That is we locate the handle $_{n\ 1}$ in $_{n\ 1}$ and reduce this handle to obtain the right sentential form $_{n\ 2}$ If by continuing this process we produce a right sentential form consisting only of the start symbol S then we halt and announce successful completion of parsing The reverse of the sequence of productions used in the reductions is a rightmost derivation for the input string

4 5 3 Shift Reduce Parsing

Shift reduce parsing is a form of bottom up parsing in which a stack holds grammar symbols and an input bu er holds the rest of the string to be parsed As we shall see the handle always appears at the top of the stack just before it is identi ed as the handle

We use to mark the bottom of the stack and also the right end of the input Conventionally when discussing bottom up parsing we show the top of the stack on the right rather than on the left as we did for top down parsing Initially the stack is empty and the string w is on the input as follows

STACK	INPUT
	w

During a left to right scan of the input string the parser shifts zero or more input symbols onto the stack until it is ready to reduce a string of grammar symbols on top of the stack It then reduces to the head of the appropriate production The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty

STACK	INPUT
S	

Upon entering this con guration the parser halts and announces successful completion of parsing Figure 4 28 steps through the actions a shift reduce parser might take in parsing the input string $\mathbf{id}_1$ $\mathbf{id}_2$ according to the expression grammar 4 1

STACK	INPUT	ACTION
	$\mathbf{id}_1$ $\mathbf{id}_2$	shift
$\mathbf{id}_1$	$\mathbf{id}_2$	reduce by F $\mathbf{id}$
F	$\mathbf{id}_2$	reduce by T F
T	$\mathbf{id}_2$	shift
T	$\mathbf{id}_2$	shift
T $\mathbf{id}_2$		reduce by F $\mathbf{id}$
T F		reduce by T T F
T		reduce by E T
E		accept

Figure 4 28 Con gurations of a shift reduce parser on input $\mathbf{id}_1$ $\mathbf{id}_2$

While the primary operations are shift and reduce there are actually four possible actions a shift reduce parser can make 1 shift 2 reduce 3 accept and 4 error

1 *Shift* Shift the next input symbol onto the top of the stack

2 *Reduce* The right end of the string to be reduced must be at the top of the stack Locate the left end of the string within the stack and decide with what nonterminal to replace the string

3 *Accept* Announce successful completion of parsing

4 *Error* Discover a syntax error and call an error recovery routine

The use of a stack in shift reduce parsing is justi ed by an important fact the handle will always eventually appear on top of the stack never inside This fact can be shown by considering the possible forms of two successive steps in any rightmost derivation Figure 4 29 illustrates the two possible cases In case 1 A is replaced by By and then the rightmost nonterminal B in the body By is replaced by In case 2 A is again expanded rst but this time the body is a string y of terminals only The next rightmost nonterminal B will be somewhere to the left of y

In other words

$$\begin{array}{ll} 1 & S \underset{rm}{\quad} Az \underset{rm}{\quad} Byz \underset{rm}{\quad} yz \\ 2 & S \underset{rm}{\quad} BxAz \underset{rm}{\quad} Bxyz \underset{rm}{\quad} xyz \end{array}$$

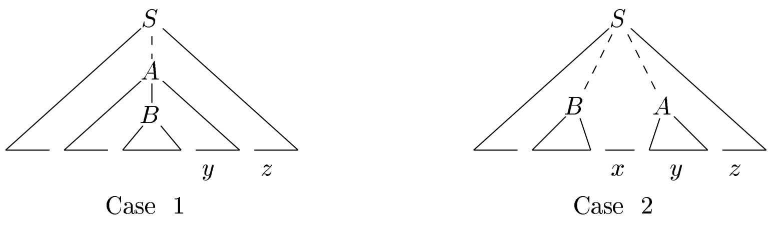

Figure 4 29 Cases for two successive steps of a rightmost derivation

Consider case 1 in reverse where a shift reduce parser has just reached the con guration

STACK	INPUT
	yz

The parser reduces the handle to B to reach the con guration

B	yz

The parser can now shift the string y onto the stack by a sequence of zero or more shift moves to reach the con guration

By	z

with the handle By on top of the stack and it gets reduced to A

Now consider case 2 In con guration

	xyz

the handle is on top of the stack After reducing the handle to B the parser can shift the string xy to get the next handle y on top of the stack ready to be reduced to A

Bxy	z

In both cases after making a reduction the parser had to shift zero or more symbols to get the next handle onto the stack It never had to go into the stack to nd the handle

4 5 4 Con icts During Shift Reduce Parsing

There are context free grammars for which shift reduce parsing cannot be used Every shift reduce parser for such a grammar can reach a con guration in which the parser knowing the entire stack and also the next k input symbols cannot decide whether to shift or to reduce a *shift reduce con ict* or cannot decide which of several reductions to make a *reduce reduce con ict* We now give some examples of syntactic constructs that give rise to such grammars Techni cally these grammars are not in the LR *k* class of grammars de ned in Section 4 7 we refer to them as non LR grammars The *k* in LR *k* refers to the number of symbols of lookahead on the input Grammars used in compiling usually fall in the LR 1 class with one symbol of lookahead at most

Example 4 38 An ambiguous grammar can never be LR For example con sider the dangling else grammar 4 14 of Section 4 3

$$\begin{array}{lll} \textit{stmt} & & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\ & | & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\ & | & \textbf{other} \end{array}$$

If we have a shift reduce parser in con guration

STACK	INPUT
if *expr* **then** *stmt*	**else**

we cannot tell whether **if** *expr* **then** *stmt* is the handle no matter what appears below it on the stack Here there is a shift reduce con ict Depending on what follows the **else** on the input it might be correct to reduce **if** *expr* **then** *stmt* to *stmt* or it might be correct to shift **else** and then to look for another *stmt* to complete the alternative **if** *expr* **then** *stmt* **else** *stmt*

Note that shift reduce parsing can be adapted to parse certain ambigu ous grammars such as the if then else grammar above If we resolve the shift reduce con ict on **else** in favor of shifting the parser will behave as we expect associating each **else** with the previous unmatched **then** We discuss parsers for such ambiguous grammars in Section 4 8 □

Another common setting for con icts occurs when we know we have a han dle but the stack contents and the next input symbol are insu cient to de termine which production should be used in a reduction The next example illustrates this situation

Example 4 39 Suppose we have a lexical analyzer that returns the token name **id** for all names regardless of their type Suppose also that our lan guage invokes procedures by giving their names with parameters surrounded by parentheses and that arrays are referenced by the same syntax Since the translation of indices in array references and parameters in procedure calls are di erent we want to use di erent productions to generate lists of actual parameters and indices Our grammar might therefore have among others productions such as those in Fig 4 30

A statement beginning with `p i j` would appear as the token stream **id id id** to the parser After shifting the rst three tokens onto the stack a shift reduce parser would be in con guration

1	*stmt*	**id** *parameter_list*
2	*stmt*	*expr* *expr*
3	*parameter_list*	*parameter_list* *parameter*
4	*parameter_list*	*parameter*
5	*parameter*	**id**
6	*expr*	**id** *expr_list*
7	*expr*	**id**
8	*expr_list*	*expr_list* *expr*
9	*expr_list*	*expr*

Figure 4 30 Productions involving procedure calls and array references

STACK	INPUT
id **id**	**id**

It is evident that the **id** on top of the stack must be reduced but by which production The correct choice is production 5 if p is a procedure but pro duction 7 if p is an array The stack does not tell which information in the symbol table obtained from the declaration of p must be used

One solution is to change the token **id** in production 1 to **procid** and to use a more sophisticated lexical analyzer that returns the token name **procid** when it recognizes a lexeme that is the name of a procedure Doing so would require the lexical analyzer to consult the symbol table before returning a token

If we made this modi cation then on processing `p i j` the parser would be either in the con guration

STACK	INPUT
procid **id**	**id**

or in the con guration above In the former case we choose reduction by production 5 in the latter case by production 7 Notice how the symbol third from the top of the stack determines the reduction to be made even though it is not involved in the reduction Shift reduce parsing can utilize information far down in the stack to guide the parse □

4 5 5 Exercises for Section 4 5

Exercise 4 5 1 For the grammar S 0 S 1 | 0 1 of Exercise 4 2 2 a indicate the handle in each of the following right sentential forms

a 000111

b 00S11

Exercise 4 5 2 Repeat Exercise 4 5 1 for the grammar S $S\,S$ | $S\,S$ | a of Exercise 4 2 1 and the following right sentential forms a SSS a

b SS a a

c aaa a

Exercise 4 5 3 Give bottom up parses for the following input strings and grammars

a The input 000111 according to the grammar of Exercise 4 5 1

b The input aaa a according to the grammar of Exercise 4 5 2

4 6 Introduction to LR Parsing Simple LR

The most prevalent type of bottom up parser today is based on a concept called LR k parsing the L is for left to right scanning of the input the R for constructing a rightmost derivation in reverse and the k for the number of input symbols of lookahead that are used in making parsing decisions The cases k 0 or k 1 are of practical interest and we shall only consider LR parsers with k 1 here When k is omitted k is assumed to be 1

This section introduces the basic concepts of LR parsing and the easiest method for constructing shift reduce parsers called simple LR or SLR for short Some familiarity with the basic concepts is helpful even if the LR parser itself is constructed using an automatic parser generator We begin with items and parser states the diagnostic output from an LR parser generator typically includes parser states which can be used to isolate the sources of parsing con icts

Section 4 7 introduces two more complex methods canonical LR and LALR that are used in the majority of LR parsers

4 6 1 Why LR Parsers

LR parsers are table driven much like the nonrecursive LL parsers of Sec tion 4 4 4 A grammar for which we can construct a parsing table using one of the methods in this section and the next is said to be an *LR grammar* Intu itively for a grammar to be LR it is su cient that a left to right shift reduce parser be able to recognize handles of right sentential forms when they appear on top of the stack

LR parsing is attractive for a variety of reasons

LR parsers can be constructed to recognize virtually all programming language constructs for which context free grammars can be written Non LR context free grammars exist but these can generally be avoided for typical programming language constructs

The LR parsing method is the most general nonbacktracking shift reduce parsing method known yet it can be implemented as e ciently as other more primitive shift reduce methods see the bibliographic notes

An LR parser can detect a syntactic error as soon as it is possible to do so on a left to right scan of the input

The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive or LL methods For a grammar to be LR k we must be able to recognize the occurrence of the right side of a production in a right sentential form with k input symbols of lookahead This requirement is far less stringent than that for LL k grammars where we must be able to recognize the use of a production seeing only the rst k symbols of what its right side derives Thus it should not be surprising that LR grammars can describe more languages than LL grammars

The principal drawback of the LR method is that it is too much work to construct an LR parser by hand for a typical programming language grammar A specialized tool an LR parser generator is needed Fortunately many such generators are available and we shall discuss one of the most commonly used ones `Yacc` in Section 4 9 Such a generator takes a context free grammar and automatically produces a parser for that grammar If the grammar contains ambiguities or other constructs that are di cult to parse in a left to right scan of the input then the parser generator locates these constructs and provides detailed diagnostic messages

4 6 2 Items and the LR 0 Automaton

How does a shift reduce parser know when to shift and when to reduce For example with stack contents T and next input symbol in Fig 4 28 how does the parser know that T on the top of the stack is not a handle so the appropriate action is to shift and not to reduce T to E

An LR parser makes shift reduce decisions by maintaining states to keep track of where we are in a parse States represent sets of items An *LR 0 item item* for short of a grammar G is a production of G with a dot at some position of the body Thus production A XYZ yields the four items

$$\begin{array}{ll} A & \;XYZ \\ A & X\;YZ \\ A & XY\;Z \\ A & XYZ \end{array}$$

The production A generates only one item A

Intuitively an item indicates how much of a production we have seen at a given point in the parsing process For example the item A XYZ indicates that we hope to see a string derivable from XYZ next on the input Item

Representing Item Sets

A parser generator that produces a bottom up parser may need to rep resent items and sets of items conveniently Note that an item can be represented by a pair of integers the rst of which is the number of one of the productions of the underlying grammar and the second of which is the position of the dot Sets of items can be represented by a list of these pairs However as we shall see the necessary sets of items often include closure items where the dot is at the beginning of the body These can always be reconstructed from the other items in the set and we do not have to include them in the list

$A \quad X\,YZ$ indicates that we have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ Item $A \quad XYZ$ indicates that we have seen the body XYZ and that it may be time to reduce XYZ to A

One collection of sets of LR 0 items called the *canonical* LR 0 collection provides the basis for constructing a deterministic nite automaton that is used to make parsing decisions Such an automaton is called an *LR 0 automaton* [3] In particular each state of the LR 0 automaton represents a set of items in the canonical LR 0 collection The automaton for the expression grammar 4 1 shown in Fig 4 31 will serve as the running example for discussing the canonical LR 0 collection for a grammar

To construct the canonical LR 0 collection for a grammar we de ne an augmented grammar and two functions CLOSURE and GOTO If G is a grammar with start symbol S then G' the *augmented grammar* for G is G with a new start symbol S' and production $S' \quad S$ The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input That is acceptance occurs when and only when the parser is about to reduce by $S' \quad S$

Closure of Item Sets

If I is a set of items for a grammar G then CLOSURE I is the set of items constructed from I by the two rules

1 Initially add every item in I to CLOSURE I

2 If $A \quad\quad B$ is in CLOSURE I and $B \quad\quad$ is a production then add the item $B \quad\quad$ to CLOSURE I if it is not already there Apply this rule until no more new items can be added to CLOSURE I

[3]Technically the automaton misses being deterministic according to the de nition of Sec tion 3 6 4 because we do not have a dead state corresponding to the empty set of items As a result there are some state input pairs for which no next state exists

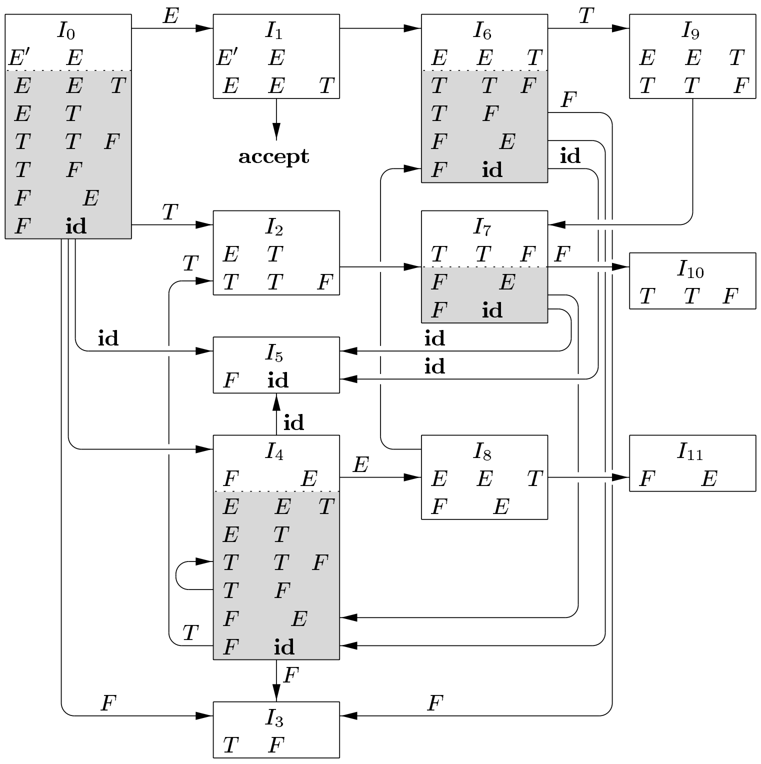

Figure 4 31 LR 0 automaton for the expression grammar 4 1

Intuitively A B in CLOSURE I indicates that at some point in the parsing process we think we might next see a substring derivable from B as input The substring derivable from B will have a pre x derivable from B by applying one of the B productions We therefore add items for all the B productions that is if B is a production we also include B in CLOSURE I

Example 4 40 Consider the augmented expression grammar

$$\begin{array}{rl} E' & E \\ E & E \quad T \mid T \\ T & T \quad F \mid F \\ F & \quad E \quad \mid \textbf{id} \end{array}$$

If I is the set of one item $\{ E' \quad E \}$ then CLOSURE I contains the set of items I_0 in Fig 4 31

To see how the closure is computed E' E is put in CLOSURE I by rule 1 Since there is an E immediately to the right of a dot we add the E productions with dots at the left ends E E T and E T Now there is a T immediately to the right of a dot in the latter item so we add T T F and T F Next the F to the right of a dot forces us to add F E and F **id** but no other items need to be added □

The closure can be computed as in Fig 4 32 A convenient way to imple ment the function *closure* is to keep a boolean array *added* indexed by the nonterminals of G such that *added* B is set to **true** if and when we add the item B for each B production B

```
SetOfItems CLOSURE $I$ {
        $J$  $I$
        repeat
                for  each item $A$      $B$  in $J$
                        for  each production $B$      of $G$
                                if  $B$       is not in $J$
                                        add $B$       to $J$
        until no more items are added to $J$ on one round
        return $J$
}
```

Figure 4 32 Computation of CLOSURE

Note that if one B production is added to the closure of I with the dot at the left end then all B productions will be similarly added to the closure Hence it is not necessary in some circumstances actually to list the items B added to I by CLOSURE A list of the nonterminals B whose productions were so added will su ce We divide all the sets of items of interest into two classes

1 *Kernel items* the initial item S' S and all items whose dots are not at the left end

2 *Nonkernel items* all items with their dots at the left end except for S' S

Moreover each set of items of interest is formed by taking the closure of a set of kernel items the items added in the closure can never be kernel items of course Thus we can represent the sets of items we are really interested in with very little storage if we throw away all nonkernel items knowing that they could be regenerated by the closure process In Fig 4 31 nonkernel items are in the shaded part of the box for a state

The Function GOTO

The second useful function is GOTO I X where I is a set of items and X is a grammar symbol GOTO I X is de ned to be the closure of the set of all items A X such that A X is in I Intuitively the GOTO function is used to de ne the transitions in the LR 0 automaton for a grammar The states of the automaton correspond to sets of items and GOTO I X speci es the transition from the state for I under input X

Example 4 41 If I is the set of two items $\{ E' \quad E \quad E \quad E \quad T \}$ then GOTO I contains the items

$$\begin{array}{ll} E & E \quad T \\ T & T \quad F \\ T & F \\ F & \quad E \\ F & \textbf{id} \end{array}$$

We computed GOTO I by examining I for items with immediately to the right of the dot E' E is not such an item but E E T is We moved the dot over the to get E E T and then took the closure of this singleton set □

We are now ready for the algorithm to construct C the canonical collection of sets of LR 0 items for an augmented grammar G' the algorithm is shown in Fig 4 33

```
void items G' {
        C     CLOSURE { S'    S }
        repeat
                for   each set of items I in C
                        for   each grammar symbol X
                                if   GOTO I X  is not empty and not in C
                                        add GOTO I X  to C
        until no new sets of items are added to C on a round
}
```

Figure 4 33 Computation of the canonical collection of sets of LR 0 items

Example 4 42 The canonical collection of sets of LR 0 items for grammar 4 1 and the GOTO function are shown in Fig 4 31 GOTO is encoded by the transitions in the gure □

Use of the LR 0 Automaton

The central idea behind Simple LR or SLR parsing is the construction from the grammar of the LR 0 automaton The states of this automaton are the sets of items from the canonical LR 0 collection and the transitions are given by the GOTO function The LR 0 automaton for the expression grammar 4 1 appeared earlier in Fig 4 31

The start state of the LR 0 automaton is CLOSURE $\{ S' \quad S \}$ where S' is the start symbol of the augmented grammar All states are accepting states We say state j to refer to the state corresponding to the set of items I_j

How can LR 0 automata help with shift reduce decisions Shift reduce decisions can be made as follows Suppose that the string of grammar symbols takes the LR 0 automaton from the start state 0 to some state j Then shift on next input symbol a if state j has a transition on a Otherwise we choose to reduce the items in state j will tell us which production to use

The LR parsing algorithm to be introduced in Section 4 6 3 uses its stack to keep track of states as well as grammar symbols in fact the grammar symbol can be recovered from the state so the stack holds states The next example gives a preview of how an LR 0 automaton and a stack of states can be used to make shift reduce parsing decisions

Example 4 43 Figure 4 34 illustrates the actions of a shift reduce parser on input **id** **id** using the LR 0 automaton in Fig 4 31 We use a stack to hold states for clarity the grammar symbols corresponding to the states on the stack appear in column SYMBOLS At line 1 the stack holds the start state 0 of the automaton the corresponding symbol is the bottom of stack marker

LINE	STACK	SYMBOLS	INPUT	ACTION
1	0		**id** **id**	shift to 5
2	0 5	**id**	**id**	reduce by F **id**
3	0 3	F	**id**	reduce by T F
4	0 2	T	**id**	shift to 7
5	0 2 7	T	**id**	shift to 5
6	0 2 7 5	T **id**		reduce by F **id**
7	0 2 7 10	T F		reduce by T T F
8	0 2	T		reduce by E T
9	0 1	E		accept

Figure 4 34 The parse of **id** **id**

The next input symbol is **id** and state 0 has a transition on **id** to state 5 We therefore shift At line 2 state 5 symbol **id** has been pushed onto the stack There is no transition from state 5 on input so we reduce From item F **id** in state 5 the reduction is by production F **id**

With symbols a reduction is implemented by popping the body of the pro duction from the stack on line 2 the body is **id** and pushing the head of the production in this case F With states we pop state 5 for symbol **id** which brings state 0 to the top and look for a transition on F the head of the production In Fig 4 31 state 0 has a transition on F to state 3 so we push state 3 with corresponding symbol F see line 3

As another example consider line 5 with state 7 symbol on top of the stack This state has a transition to state 5 on input **id** so we push state 5 symbol **id** State 5 has no transitions so we reduce by F **id** When we pop state 5 for the body **id** state 7 comes to the top of the stack Since state 7 has a transition on F to state 10 we push state 10 symbol F □

4 6 3 The LR Parsing Algorithm

A schematic of an LR parser is shown in Fig 4 35 It consists of an input an output a stack a driver program and a parsing table that has two parts ACTION and GOTO The driver program is the same for all LR parsers only the parsing table changes from one parser to another The parsing program reads characters from an input bu er one at a time Where a shift reduce parser would shift a symbol an LR parser shifts a *state* Each state summarizes the information contained in the stack below it

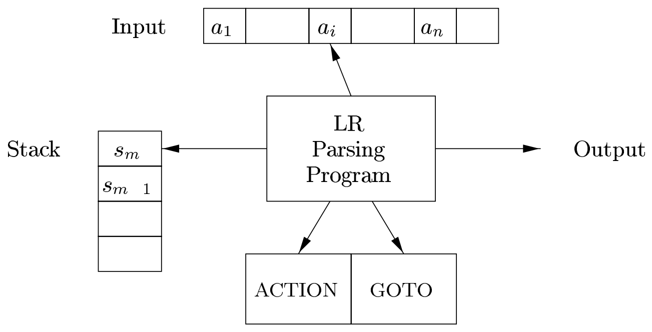

Figure 4 35 Model of an LR parser

The stack holds a sequence of states $s_0 s_1$ s_m where s_m is on top In the SLR method the stack holds states from the LR 0 automaton the canonical LR and LALR methods are similar By construction each state has a corre sponding grammar symbol Recall that states correspond to sets of items and that there is a transition from state i to state j if GOTO I_i X I_j All tran sitions to state j must be for the same grammar symbol X Thus each state except the start state 0 has a unique grammar symbol associated with it [4]

[4]The converse need not hold that is more than one state may have the same grammar

Structure of the LR Parsing Table

The parsing table consists of two parts a parsing action function ACTION and a goto function GOTO

1 The ACTION function takes as arguments a state i and a terminal a or the input endmarker The value of ACTION i a can have one of four forms

   a Shift j where j is a state The action taken by the parser e ectively shifts input a to the stack but uses state j to represent a

   b Reduce A The action of the parser e ectively reduces on the top of the stack to head A

   c Accept The parser accepts the input and nishes parsing

   d Error The parser discovers an error in its input and takes some corrective action We shall have more to say about how such error recovery routines work in Sections 4 8 3 and 4 9 4

2 We extend the GOTO function de ned on sets of items to states if GOTO I_i A I_j then GOTO also maps a state i and a nonterminal A to state j

LR Parser Con gurations

To describe the behavior of an LR parser it helps to have a notation repre senting the complete state of the parser its stack and the remaining input A *con guration* of an LR parser is a pair

$$s_0 s_1 \quad s_m \; a_i a_{i \;\; 1} \quad a_n$$

where the rst component is the stack contents top on the right and the second component is the remaining input This con guration represents the right sentential form

$$X_1 X_2 \quad X_m a_i a_{i \;\; 1} \quad a_n$$

in essentially the same way as a shift reduce parser would the only di erence is that instead of grammar symbols the stack holds states from which grammar symbols can be recovered That is X_i is the grammar symbol represented by state s_i Note that s_0 the start state of the parser does not represent a grammar symbol and serves as a bottom of stack marker as well as playing an important role in the parse

symbol See for example states 1 and 8 in the LR 0 automaton in Fig 4 31 which are both entered by transitions on E or states 2 and 9 which are both entered by transitions on T

Behavior of the LR Parser

The next move of the parser from the con guration above is determined by reading a_i the current input symbol and s_m the state on top of the stack and then consulting the entry ACTION s_m a_i in the parsing action table The con gurations resulting after each of the four types of move are as follows

1 If ACTION s_m a_i shift s the parser executes a shift move it shifts the next state s onto the stack entering the con guration

$$s_0 s_1 \quad s_m s \quad a_{i \ 1} \quad a_n$$

The symbol a_i need not be held on the stack since it can be recovered from s if needed which in practice it never is The current input symbol is now $a_{i \ 1}$

2 If ACTION s_m a_i reduce A then the parser executes a reduce move entering the con guration

$$s_0 s_1 \quad s_{m \ r} s \quad a_i a_{i \ 1} \quad a_n$$

where r is the length of and s GOTO $s_{m \ r}$ A Here the parser rst popped r state symbols o the stack exposing state $s_{m \ r}$ The parser then pushed s the entry for GOTO $s_{m \ r}$ A onto the stack The current input symbol is not changed in a reduce move For the LR parsers we shall construct $X_{m \ r \ 1}$ X_m the sequence of grammar symbols corresponding to the states popped o the stack will always match the right side of the reducing production

The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production For the time being we shall assume the output consists of just printing the reducing production

3 If ACTION s_m a_i accept parsing is completed

4 If ACTION s_m a_i error the parser has discovered an error and calls an error recovery routine

The LR parsing algorithm is summarized below All LR parsers behave in this fashion the only di erence between one LR parser and another is the information in the ACTION and GOTO elds of the parsing table

Algorithm 4 44 LR parsing algorithm

INPUT An input string w and an LR parsing table with functions ACTION and GOTO for a grammar G

OUTPUT If w is in L G the reduction steps of a bottom up parse for w otherwise an error indication

METHOD Initially the parser has s_0 on its stack where s_0 is the initial state and w in the input bu er The parser then executes the program in Fig 4 36 □

```
let $a$ be the  rst symbol of $w$
while 1 {    repeat forever
        let $s$ be the state on top of the stack
        if  ACTION $s$ $a$    shift $t$  {
                push $t$ onto the stack
                let $a$ be the next input symbol
        } else if  ACTION $s$ $a$    reduce $A$         {
                pop |  | symbols o  the stack
                let state $t$ now be on top of the stack
                push GOTO $t$ $A$ onto the stack
                output the production $A$
        } else if  ACTION $s$ $a$    accept   break      parsing is done
        else call error recovery routine
}
```

Figure 4 36 LR parsing program

Example 4 45 Figure 4 37 shows the ACTION and GOTO functions of an LR parsing table for the expression grammar 4 1 repeated here with the productions numbered

1	E E T	4	T F
2	E T	5	F E
3	T T F	6	F **id**

The codes for the actions are

1 si means shift and stack state i

2 rj means reduce by the production numbered j

3 acc means accept

4 blank means error

Note that the value of GOTO s a for terminal a is found in the ACTION eld connected with the shift action on input a for state s The GOTO eld gives GOTO s A for nonterminals A Although we have not yet explained how the entries for Fig 4 37 were selected we shall deal with this issue shortly

STATE	ACTION						GOTO		
	id						*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Figure 4 37 Parsing table for expression grammar

On input **id** **id** **id** the sequence of stack and input contents is shown in Fig 4 38 Also shown for clarity are the sequences of grammar symbols corresponding to the states held on the stack For example at line 1 the LR parser is in state 0 the initial state with no grammar symbol and with **id** the rst input symbol The action in row 0 and column **id** of the action eld of Fig 4 37 is s5 meaning shift by pushing state 5 That is what has happened at line 2 the state symbol 5 has been pushed onto the stack and **id** has been removed from the input

Then becomes the current input symbol and the action of state 5 on input is to reduce by F **id** One state symbol is popped o the stack State 0 is then exposed Since the goto of state 0 on F is 3 state 3 is pushed onto the stack We now have the con guration in line 3 Each of the remaining moves is determined similarly □

4 6 4 Constructing SLR Parsing Tables

The SLR method for constructing parsing tables is a good starting point for studying LR parsing We shall refer to the parsing table constructed by this method as an SLR table and to an LR parser using an SLR parsing table as an SLR parser The other two methods augment the SLR method with lookahead information

The SLR method begins with LR 0 items and LR 0 automata introduced in Section 4 5 That is given a grammar G we augment G to produce G' with a new start symbol S' From G' we construct C the canonical collection of sets of items for G' together with the GOTO function

	STACK	SYMBOLS	INPUT	ACTION
1	0		**id id id**	shift
2	0 5	**id**	**id id**	reduce by F **id**
3	0 3	F	**id id**	reduce by T F
4	0 2	T	**id id**	shift
5	0 2 7	T	**id id**	shift
6	0 2 7 5	T **id**	**id**	reduce by F **id**
7	0 2 7 10	T F	**id**	reduce by T T F
8	0 2	T	**id**	reduce by E T
9	0 1	E	**id**	shift
10	0 1 6	E	**id**	shift
11	0 1 6 5	E **id**		reduce by F **id**
12	0 1 6 3	E F		reduce by T F
13	0 1 6 9	E T		reduce by E E T
14	0 1	E		accept

Figure 4 38 Moves of an LR parser on **id id id**

The ACTION and GOTO entries in the parsing table are then constructed using the following algorithm It requires us to know FOLLOW A for each nonterminal A of a grammar see Section 4 4

Algorithm 4 46 Constructing an SLR parsing table

INPUT An augmented grammar G'

OUTPUT The SLR parsing table functions ACTION and GOTO for G'

METHOD

1 Construct C $\{I_0\ I_1\ \ \ I_n\}$ the collection of sets of LR 0 items for G'

2 State i is constructed from I_i The parsing actions for state i are deter mined as follows

   a If A a is in I_i and GOTO I_i a I_j then set ACTION i a to shift j Here a must be a terminal

   b If A is in I_i then set ACTION i a to reduce A for all a in FOLLOW A here A may not be S'

   c If S' S is in I_i then set ACTION i to accept

   If any con icting actions result from the above rules we say the grammar is not SLR 1 The algorithm fails to produce a parser in this case

3 The goto transitions for state i are constructed for all nonterminals A using the rule If GOTO I_i A I_j then GOTO i A j

4 All entries not de ned by rules 2 and 3 are made error

5 The initial state of the parser is the one constructed from the set of items containing S' S

□

The parsing table consisting of the ACTION and GOTO functions determined by Algorithm 4 46 is called the *SLR 1 table for G* An LR parser using the SLR 1 table for G is called the SLR 1 parser for G and a grammar having an SLR 1 parsing table is said to be *SLR 1* We usually omit the 1 after the SLR since we shall not deal here with parsers having more than one symbol of lookahead

Example 4 47 Let us construct the SLR table for the augmented expression grammar The canonical collection of sets of LR 0 items for the grammar was shown in Fig 4 31 First consider the set of items I_0

$$\begin{array}{lll}
E' & & E \\
E & & E \quad T \\
E & & T \\
T & & T \quad F \\
T & & F \\
F & & \quad E \\
F & & \textbf{id}
\end{array}$$

The item F E gives rise to the entry ACTION 0 shift 4 and the item F **id** to the entry ACTION 0 **id** shift 5 Other items in I_0 yield no actions Now consider I_1

$$\begin{array}{lll}
E' & & E \\
E & & E \quad T
\end{array}$$

The rst item yields ACTION 1 accept and the second yields ACTION 1 shift 6 Next consider I_2

$$\begin{array}{lll}
E & & T \\
T & & T \quad F
\end{array}$$

Since FOLLOW E $\{$ $\}$ the rst item makes

ACTION 2 ACTION 2 ACTION 2 reduce E T

The second item makes ACTION 2 shift 7 Continuing in this fashion we obtain the ACTION and GOTO tables that were shown in Fig 4 31 In that gure the numbers of productions in reduce actions are the same as the order in which they appear in the original grammar 4 1 That is E E T is number 1 E T is 2 and so on □

Example 4 48 Every SLR 1 grammar is unambiguous but there are many unambiguous grammars that are not SLR 1 Consider the grammar with pro ductions

$$\begin{array}{lll} S & & L \quad R \mid R \\ L & & \quad R \mid \mathbf{id} \\ R & & L \end{array} \qquad 4\ 49$$

Think of L and R as standing for l value and r value respectively and as an operator indicating contents of [5] The canonical collection of sets of LR 0 items for grammar 4 49 is shown in Fig 4 39

I_0	$S' \quad S$	I_5	$L \quad \mathbf{id}$
	$S \quad L \quad R$		
	$S \quad R$	I_6	$S \quad L \quad R$
	$L \quad\quad R$		$R \quad L$
	$L \quad \mathbf{id}$		$L \quad\quad R$
	$R \quad L$		$L \quad \mathbf{id}$
I_1	$S' \quad S$	I_7	$L \quad\quad R$
I_2	$S \quad L \quad R$	I_8	$R \quad L$
	$R \quad L$		
		I_9	$S \quad L \quad R$
I_3	$S \quad R$		
I_4	$L \quad\quad R$		
	$R \quad L$		
	$L \quad\quad R$		
	$L \quad \mathbf{id}$		

Figure 4 39 Canonical LR 0 collection for grammar 4 49

Consider the set of items I_2 The rst item in this set makes ACTION 2 be shift 6 Since FOLLOW R contains to see why consider the derivation $S \quad L \quad R \qquad R \quad R$ the second item sets ACTION 2 to reduce $R \quad L$ Since there is both a shift and a reduce entry in ACTION 2 state 2 has a shift reduce con ict on input symbol

Grammar 4 49 is not ambiguous This shift reduce con ict arises from the fact that the SLR parser construction method is not powerful enough to remember enough left context to decide what action the parser should take on input having seen a string reducible to L The canonical and LALR methods to be discussed next will succeed on a larger collection of grammars including

[5] As in Section 2 8 3 an l value designates a location and an r value is a value that can be stored in a location

grammar 4 49 Note however that there are unambiguous grammars for which every LR parser construction method will produce a parsing action table with parsing action con icts Fortunately such grammars can generally be avoided in programming language applications □

4 6 5 Viable Pre xes

Why can LR 0 automata be used to make shift reduce decisions The LR 0 automaton for a grammar characterizes the strings of grammar symbols that can appear on the stack of a shift reduce parser for the grammar The stack contents must be a pre x of a right sentential form If the stack holds and the rest of the input is x then a sequence of reductions will take x to S In terms of derivations $S \;\underset{rm}{}\; x$

Not all pre xes of right sentential forms can appear on the stack however since the parser must not shift past the handle For example suppose

$$E \;\underset{rm}{}\; F \;\mathbf{id}\; \underset{rm}{}\; E \;\;\mathbf{id}$$

Then at various times during the parse the stack will hold E and E but it must not hold E since E is a handle which the parser must reduce to F before shifting

The pre xes of right sentential forms that can appear on the stack of a shift reduce parser are called *viable pre xes* They are de ned as follows a viable pre x is a pre x of a right sentential form that does not continue past the right end of the rightmost handle of that sentential form By this de nition it is always possible to add terminal symbols to the end of a viable pre x to obtain a right sentential form

SLR parsing is based on the fact that LR 0 automata recognize viable pre xes We say item $A \quad _1 \; _2$ is *valid* for a viable pre x $\; _1$ if there is a derivation $S' \;\underset{rm}{}\; Aw \;\underset{rm}{}\; \; _1 \; _2 w$ In general an item will be valid for many viable pre xes

The fact that $A \quad _1 \; _2$ is valid for $\; _1$ tells us a lot about whether to shift or reduce when we nd $\; _1$ on the parsing stack In particular if $_2$ / then it suggests that we have not yet shifted the handle onto the stack so shift is our move If $_2$ then it looks as if $A \quad _1$ is the handle and we should reduce by this production Of course two valid items may tell us to do di erent things for the same viable pre x Some of these con icts can be resolved by looking at the next input symbol and others can be resolved by the methods of Section 4 8 but we should not suppose that all parsing action con icts can be resolved if the LR method is applied to an arbitrary grammar

We can easily compute the set of valid items for each viable pre x that can appear on the stack of an LR parser In fact it is a central theorem of LR parsing theory that the set of valid items for a viable pre x is exactly the set of items reached from the initial state along the path labeled in the LR 0 automaton for the grammar In essence the set of valid items embodies

Items as States of an NFA

A nondeterministic nite automaton N for recognizing viable pre xes can be constructed by treating the items themselves as states There is a transition from A X to A X labeled X and there is a transition from A B to B labeled Then CLOSURE I for set of items states of N I is exactly the closure of a set of NFA states de ned in Section 3 7 1 Thus GOTO I X gives the transition from I on symbol X in the DFA constructed from N by the subset construction Viewed in this way the procedure *items* G' in Fig 4 33 is just the subset construction itself applied to the NFA N with items as states

all the useful information that can be gleaned from the stack While we shall not prove this theorem here we shall give an example

Example 4 50 Let us consider the augmented expression grammar again whose sets of items and GOTO function are exhibited in Fig 4 31 Clearly the string E T is a viable pre x of the grammar The automaton of Fig 4 31 will be in state 7 after having read E T State 7 contains the items

$$\begin{array}{lll} T & T & F \\ F & & E \\ F & & \textbf{id} \end{array}$$

which are precisely the items valid for E T To see why consider the following three rightmost derivations

$$\begin{array}{lllll} E' & E & \qquad & E' & E \\ \;_{rm} & E \;\; T & & \;_{rm} & E \;\; T \\ \;_{rm} & E \;\; T \;\; F & & \;_{rm} & E \;\; T \;\; F \\ \;_{rm} & & & \;_{rm} & E \;\; T \;\; E \\ & & & \;_{rm} & \end{array} \qquad \begin{array}{ll} E' & E \\ \;_{rm} & E \;\; T \\ \;_{rm} & E \;\; T \;\; F \\ \;_{rm} & E \;\; T \;\; \textbf{id} \\ \;_{rm} & \end{array}$$

The rst derivation shows the validity of T T F the second the validity of F E and the third the validity of F **id** It can be shown that there are no other valid items for E T although we shall not prove that fact here □

4 6 6 Exercises for Section 4 6

Exercise 4 6 1 Describe all the viable pre xes for the following grammars

a The grammar S $0\,S\,1 \mid 0\,1$ of Exercise 4 2 2 a b The grammar S $S\ S$ $|\ S\ S$ $|\ a$ of Exercise 4 2 1

c The grammar S S S $|$ of Exercise 4 2 2 c

Exercise 4 6 2 Construct the SLR sets of items for the augmented grammar of Exercise 4 2 1 Compute the GOTO function for these sets of items Show the parsing table for this grammar Is the grammar SLR

Exercise 4 6 3 Show the actions of your parsing table from Exercise 4 6 2 on the input aa a

Exercise 4 6 4 For each of the augmented grammars of Exercise 4 2 2 a g

a Construct the SLR sets of items and their GOTO function

b Indicate any action con icts in your sets of items

c Construct the SLR parsing table if one exists

Exercise 4 6 5 Show that the following grammar

$$\begin{array}{ll} S & A\ a\ A\ b\ |\ B\ b\ B\ a \\ A & \\ B & \end{array}$$

is LL 1 but not SLR 1

Exercise 4 6 6 Show that the following grammar

$$\begin{array}{ll} S & S\ A\ |\ A \\ A & a \end{array}$$

is SLR 1 but not LL 1

Exercise 4 6 7 Consider the family of grammars G_n de ned by

$$\begin{array}{llll} S & A_i\ b_i & \text{for } 1 & i \quad n \\ A_i & a_j\ A_i\ |\ a_j & \text{for } 1 & i\ j \quad n \text{ and } i\ /\ j \end{array}$$

Show that

a G_n has $2n^2$ n productions

b G_n has 2^n n^2 n sets of LR 0 items

c G_n is SLR 1

What does this analysis say about how large LR parsers can get

Exercise 4 6 8 We suggested that individual items could be regarded as states of a nondeterministic nite automaton while sets of valid items are the states of a deterministic nite automaton see the box on Items as States of an NFA in Section 4 6 5 For the grammar S $S\ S$ $|\ S\ S$ $|\ a$ of Exercise 4 2 1

a Draw the transition diagram NFA for the valid items of this grammar according to the rule given in the box cited above

b Apply the subset construction Algorithm 3 20 to your NFA from part a How does the resulting DFA compare to the set of LR 0 items for the grammar

c Show that in all cases the subset construction applied to the NFA that comes from the valid items for a grammar produces the LR 0 sets of items

Exercise 4 6 9 The following is an ambiguous grammar

$$\begin{array}{ll} S & A\ S \mid b \\ A & S\ A \mid a \end{array}$$

Construct for this grammar its collection of sets of LR 0 items If we try to build an LR parsing table for the grammar there are certain con icting actions What are they Suppose we tried to use the parsing table by nondeterminis tically choosing a possible action whenever there is a con ict Show all the possible sequences of actions on input *abab*

4 7 More Powerful LR Parsers

In this section we shall extend the previous LR parsing techniques to use one symbol of lookahead on the input There are two di erent methods

1 The canonical LR or just LR method which makes full use of the lookahead symbol s This method uses a large set of items called the LR 1 items

2 The lookahead LR or LALR method which is based on the LR 0 sets of items and has many fewer states than typical parsers based on the LR 1 items By carefully introducing lookaheads into the LR 0 items we can handle many more grammars with the LALR method than with the SLR method and build parsing tables that are no bigger than the SLR tables LALR is the method of choice in most situations

After introducing both these methods we conclude with a discussion of how to compact LR parsing tables for environments with limited memory

4 7 1 Canonical LR 1 Items

We shall now present the most general technique for constructing an LR parsing table from a grammar Recall that in the SLR method state i calls for reduction by A if the set of items I_i contains item A and input symbol a is in FOLLOW A In some situations however when state i appears on top of the stack the viable pre x on the stack is such that A cannot be followed by a in any right sentential form Thus the reduction by A should be invalid on input a

Example 4 51 Let us reconsider Example 4 48 where in state 2 we had item R L which could correspond to A above and a could be the sign which is in FOLLOW R Thus the SLR parser calls for reduction by R L in state 2 with as the next input the shift action is also called for because of item S L R in state 2 However there is no right sentential form of the grammar in Example 4 48 that begins R Thus state 2 which is the state corresponding to viable pre x L only should not really call for reduction of that L to R □

It is possible to carry more information in the state that will allow us to rule out some of these invalid reductions by A By splitting states when necessary we can arrange to have each state of an LR parser indicate exactly which input symbols can follow a handle for which there is a possible reduction to A

The extra information is incorporated into the state by rede ning items to include a terminal symbol as a second component The general form of an item becomes A a where A is a production and a is a terminal or the right endmarker We call such an object an *LR 1 item* The 1 refers to the length of the second component called the *lookahead* of the item[6] The lookahead has no e ect in an item of the form A a where is not but an item of the form A a calls for a reduction by A only if the next input symbol is a Thus we are compelled to reduce by A only on those input symbols a for which A a is an LR 1 item in the state on top of the stack The set of such a s will always be a subset of FOLLOW A but it could be a proper subset as in Example 4 51

Formally we say LR 1 item A a is *valid* for a viable pre x if there is a derivation $S \underset{rm}{} Aw \underset{rm}{} w$ where

1 and

2 Either a is the rst symbol of w or w is and a is

Example 4 52 Let us consider the grammar

[6]Lookaheads that are strings of length greater than one are possible of course but we shall not consider such lookaheads here

$$\begin{array}{ll} S & B\,B \\ B & a\,B \mid b \end{array}$$

There is a rightmost derivation $S \underset{rm}{\quad} aaBab \underset{rm}{\quad} aaaBab$ We see that item B $a\,B\;\; a$ is valid for a viable pre x aaa by letting aa A B w ab a and B in the above de nition There is also a rightmost derivation $S \underset{rm}{\quad} BaB \underset{rm}{\quad} BaaB$ From this derivation we see that item B $a\,B$ is valid for viable pre x Baa □

4 7 2   Constructing LR 1 Sets of Items

The method for building the collection of sets of valid LR 1 items is essentially the same as the one for building the canonical collection of sets of LR 0 items We need only to modify the two procedures CLOSURE and GOTO

```
SetOfItems CLOSURE $I$ {
      repeat
            for  each item $A$        $B$  $a$ in $I$
                  for  each production $B$       in $G'$
                        for  each terminal $b$ in FIRST   $a$
                              add $B$         $b$ to set $I$
      until no more items are added to $I$
      return $I$
}

SetOfItems GOTO $I$ $X$ {
      initialize $J$ to be the empty set
      for  each item $A$       $X$   $a$ in $I$
            add item $A$       $X$   $a$ to set $J$
      return CLOSURE $J$
}

void items $G'$ {
      initialize $C$ to  CLOSURE { $S'$      $S$   }
      repeat
            for  each set of items $I$ in $C$
                  for  each grammar symbol $X$
                        if  GOTO $I$ $X$ is not empty and not in $C$
                              add GOTO $I$ $X$ to $C$
      until no new sets of items are added to $C$
}
```

Figure 4 40 Sets of LR 1 items construction for grammar G'

To appreciate the new de nition of the CLOSURE operation in particular why b must be in FIRST a consider an item of the form A B a in the set of items valid for some viable pre x Then there is a rightmost derivation S $\underset{rm}{}$ Aax $\underset{rm}{}$ B ax where Suppose ax derives terminal string by Then for each production of the form B for some we have derivation S $\underset{rm}{}$ Bby $\underset{rm}{}$ by Thus B b is valid for Note that b can be the rst terminal derived from or it is possible that derives in the derivation ax $\underset{rm}{}$ by and b can therefore be a To summarize both possibilities we say that b can be any terminal in FIRST ax where FIRST is the function from Section 4 4 Note that x cannot contain the rst terminal of by so FIRST ax FIRST a We now give the LR 1 sets of items construction

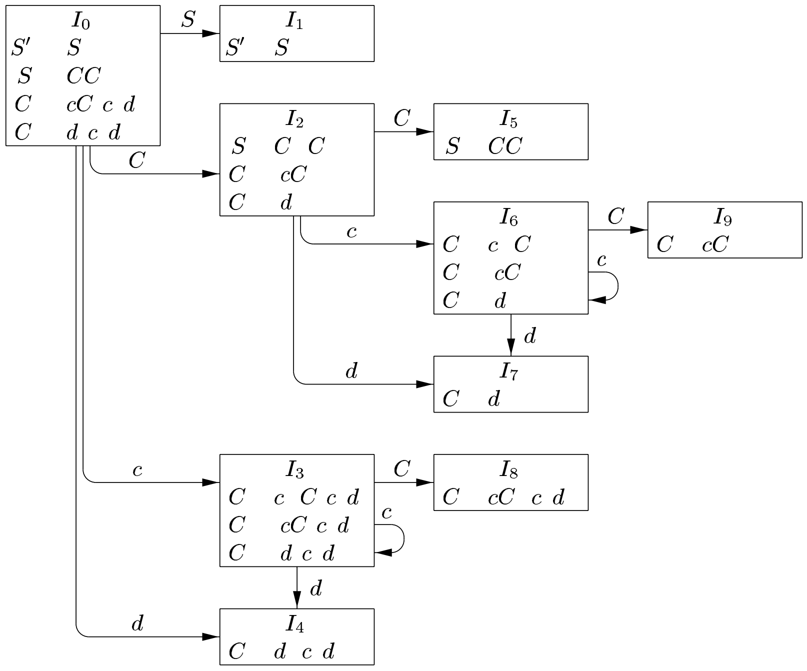

Figure 4 41 The GOTO graph for grammar 4 55

Algorithm 4 53 Construction of the sets of LR 1 items

INPUT An augmented grammar G'

OUTPUT The sets of LR 1 items that are the set of items valid for one or more viable pre xes of G'

METHOD The procedures CLOSURE and GOTO and the main routine *items* for constructing the sets of items were shown in Fig 4 40 □

Example 4 54 Consider the following augmented grammar

$$\begin{array}{lll} S' & S & \\ S & C\,C & \qquad\qquad 4\ 55 \\ C & c\,C \mid d & \end{array}$$

We begin by computing the closure of { S' S } To close we match the item S' S with the item A B a in the procedure CLOSURE That is A S' B S and a Function CLOSURE tells us to add B b for each production B and terminal b in FIRST a In terms of the present grammar B must be S CC and since is and a is b may only be Thus we add S CC

We continue to compute the closure by adding all items C b for b in FIRST C That is matching S CC against A B a we have A S B C C and a Since C does not derive the empty string FIRST C FIRST C Since FIRST C contains terminals c and d we add items C $cC\ c$ C $cC\ d$ C $d\ c$ and C $d\ d$ None of the new items has a nonterminal immediately to the right of the dot so we have completed our rst set of LR 1 items The initial set of items is

$$\begin{array}{lll} I_0 & S & S \\ & S & CC \\ & C & cC\ \ c\ d \\ & C & d\ \ c\ d \end{array}$$

The brackets have been omitted for notational convenience and we use the notation C $cC\ \ c\ d$ as a shorthand for the two items C $cC\ \ c$ and C $cC\ \ d$

Now we compute GOTO $I_0\ X$ for the various values of X For X S we must close the item S' S No additional closure is possible since the dot is at the right end Thus we have the next set of items

$$I_1 \quad S' \quad S$$

For X C we close S $C\,C$ We add the C productions with second component and then can add no more yielding

$$\begin{array}{lll} I_2 & S & C\,C \\ & C & cC \\ & C & d \end{array}$$

Next let X c We must close { C $c\,C\ \ c\ d$ } We add the C productions with second component $c\ d$ yielding

$$\begin{array}{lll} I_3 & C & c\,C\ \ c\ d \\ & C & cC\ \ c\ d \\ & C & d\ \ c\ d \end{array}$$

Finally let X d and we wind up with the set of items

$$I_4 \quad C \quad d \ \ c\ d$$

We have nished considering GOTO on I_0 We get no new sets from I_1 but I_2 has goto s on C c and d For GOTO I_2 C we get

$$I_5 \quad S \quad CC$$

no closure being needed To compute GOTO I_2 c we take the closure of $\{\, C \quad c\,C \quad \}$ to obtain

$$\begin{array}{lll} I_6 & C & c\,C \\ & C & cC \\ & C & d \end{array}$$

Note that I_6 di ers from I_3 only in second components We shall see that it is common for several sets of LR 1 items for a grammar to have the same rst components and di er in their second components When we construct the collection of sets of LR 0 items for the same grammar each set of LR 0 items will coincide with the set of rst components of one or more sets of LR 1 items We shall have more to say about this phenomenon when we discuss LALR parsing

Continuing with the GOTO function for I_2 GOTO I_2 d is seen to be

$$I_7 \quad C \quad d$$

Turning now to I_3 the GOTO s of I_3 on c and d are I_3 and I_4 respectively and GOTO I_3 C is

$$I_8 \quad C \quad cC \ \ c\ d$$

I_4 and I_5 have no GOTO s since all items have their dots at the right end The GOTO s of I_6 on c and d are I_6 and I_7 respectively and GOTO I_6 C is

$$I_9 \quad C \quad cC$$

The remaining sets of items yield no GOTO s so we are done Figure 4 41 shows the ten sets of items with their goto s □

4 7 3 Canonical LR 1 Parsing Tables

We now give the rules for constructing the LR 1 ACTION and GOTO functions from the sets of LR 1 items These functions are represented by a table as before The only di erence is in the values of the entries

Algorithm 4 56 Construction of canonical LR parsing tables

INPUT An augmented grammar G'

OUTPUT The canonical LR parsing table functions ACTION and GOTO for G'

METHOD

1 Construct C' $\{I_0$ I_1 $I_n\}$ the collection of sets of LR 1 items for G'

2 State i of the parser is constructed from I_i The parsing action for state i is determined as follows

   a If A a b is in I_i and GOTO I_i a I_j then set ACTION i a to shift j Here a must be a terminal

   b If A a is in I_i A / S' then set ACTION i a to reduce A

   c If S' S is in I_i then set ACTION i to accept

   If any con icting actions result from the above rules we say the grammar is not LR 1 The algorithm fails to produce a parser in this case

3 The goto transitions for state i are constructed for all nonterminals A using the rule If GOTO I_i A I_j then GOTO i A j

4 All entries not de ned by rules 2 and 3 are made error

5 The initial state of the parser is the one constructed from the set of items containing S' S

□

The table formed from the parsing action and goto functions produced by Algorithm 4 56 is called the *canonical* LR 1 parsing table An LR parser using this table is called a canonical LR 1 parser If the parsing action function has no multiply de ned entries then the given grammar is called an *LR 1 grammar* As before we omit the 1 if it is understood

Example 4 57 The canonical parsing table for grammar 4 55 is shown in Fig 4 42 Productions 1 2 and 3 are S CC C cC and C d respectively □

Every SLR 1 grammar is an LR 1 grammar but for an SLR 1 grammar the canonical LR parser may have more states than the SLR parser for the same grammar The grammar of the previous examples is SLR and has an SLR parser with seven states compared with the ten of Fig 4 42

STATE	ACTION			GOTO	
	c	d		S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Figure 4 42 Canonical parsing table for grammar 4 55

4 7 4   Constructing LALR Parsing Tables

We now introduce our last parser construction method the LALR *lookahead* LR technique This method is often used in practice because the tables ob tained by it are considerably smaller than the canonical LR tables yet most common syntactic constructs of programming languages can be expressed con veniently by an LALR grammar The same is almost true for SLR grammars but there are a few constructs that cannot be conveniently handled by SLR techniques see Example 4 48 for example

For a comparison of parser size the SLR and LALR tables for a grammar always have the same number of states and this number is typically several hundred states for a language like C The canonical LR table would typically have several thousand states for the same size language Thus it is much easier and more economical to construct SLR and LALR tables than the canonical LR tables

By way of introduction let us again consider grammar 4 55   whose sets of LR 1 items were shown in Fig 4 41 Take a pair of similar looking states such as I_4 and I_7 Each of these states has only items with rst component C d In I_4 the lookaheads are c or d in I_7 is the only lookahead

To see the di erence between the roles of I_4 and I_7 in the parser note that the grammar generates the regular language **c dc d** When reading an input cc $cdcc$ cd the parser shifts the rst group of c s and their following d onto the stack entering state 4 after reading the d The parser then calls for a reduction by C d provided the next input symbol is c or d The requirement that c or d follow makes sense since these are the symbols that could begin strings in **c d** If   follows the rst d we have an input like ccd which is not in the language and state 4 correctly declares an error if   is the next input

The parser enters state 7 after reading the second d Then the parser must see on the input or it started with a string not of the form **c dc d** It thus makes sense that state 7 should reduce by C d on input and declare error on inputs c or d

Let us now replace I_4 and I_7 by I_{47} the union of I_4 and I_7 consisting of the set of three items represented by C d c d The goto s on d to I_4 or I_7 from I_0 I_2 I_3 and I_6 now enter I_{47} The action of state 47 is to reduce on any input The revised parser behaves essentially like the original although it might reduce d to C in circumstances where the original would declare error for example on input like *ccd* or *cdcdc* The error will eventually be caught in fact it will be caught before any more input symbols are shifted

More generally we can look for sets of LR 1 items having the same *core* that is set of rst components and we may merge these sets with common cores into one set of items For example in Fig 4 41 I_4 and I_7 form such a pair with core $\{C \quad d\ \}$ Similarly I_3 and I_6 form another pair with core $\{C \quad c\ C\ C \quad cC\ C \quad d\}$ There is one more pair I_8 and I_9 with common core $\{C \quad cC\ \}$ Note that in general a core is a set of LR 0 items for the grammar at hand and that an LR 1 grammar may produce more than two sets of items with the same core

Since the core of GOTO I X depends only on the core of I the goto s of merged sets can themselves be merged Thus there is no problem revising the goto function as we merge sets of items The action functions are modi ed to re ect the non error actions of all sets of items in the merger

Suppose we have an LR 1 grammar that is one whose sets of LR 1 items produce no parsing action con icts If we replace all states having the same core with their union it is possible that the resulting union will have a con ict but it is unlikely for the following reason Suppose in the union there is a con ict on lookahead a because there is an item A a calling for a reduction by A and there is another item B a b calling for a shift Then some set of items from which the union was formed has item A a and since the cores of all these states are the same it must have an item B a c for some c But then this state has the same shift reduce con ict on a and the grammar was not LR 1 as we assumed Thus the merging of states with common cores can never produce a shift reduce con ict that was not present in one of the original states because shift actions depend only on the core not the lookahead

It is possible however that a merger will produce a reduce reduce con ict as the following example shows

Example 4 58 Consider the grammar

$$\begin{array}{rl} S' & S \\ S & a\,A\,d \mid b\,B\,d \mid a\,B\,e \mid b\,A\,e \\ A & c \\ B & c \end{array}$$

which generates the four strings *acd* *ace* *bcd* and *bce* The reader can check that the grammar is LR 1 by constructing the sets of items Upon doing so we nd the set of items $\{ A \quad c \quad d \quad B \quad c \quad e \}$ valid for viable pre x ac and $\{ A \quad c \ e \quad B \quad c \quad d \}$ valid for bc Neither of these sets has a con ict and their cores are the same However their union which is

$$\begin{array}{l} A \quad c \ \ d \ e \\ B \quad c \ \ d \ e \end{array}$$

generates a reduce reduce con ict since reductions by both $A \quad c$ and $B \quad c$ are called for on inputs d and e □

We are now prepared to give the rst of two LALR table construction al gorithms The general idea is to construct the sets of LR 1 items and if no con icts arise merge sets with common cores We then construct the parsing table from the collection of merged sets of items The method we are about to describe serves primarily as a de nition of LALR 1 grammars Constructing the entire collection of LR 1 sets of items requires too much space and time to be useful in practice

Algorithm 4 59 An easy but space consuming LALR table construction

INPUT An augmented grammar G'

OUTPUT The LALR parsing table functions ACTION and GOTO for G'

METHOD

1 Construct $C \quad \{I_0 \ I_1 \quad I_n\}$ the collection of sets of LR 1 items

2 For each core present among the set of LR 1 items nd all sets having that core and replace these sets by their union

3 Let $C' \quad \{J_0 \ J_1 \quad J_m\}$ be the resulting sets of LR 1 items The parsing actions for state i are constructed from J_i in the same manner as in Algorithm 4 56 If there is a parsing action con ict the algorithm fails to produce a parser and the grammar is said not to be LALR 1

4 The GOTO table is constructed as follows If J is the union of one or more sets of LR 1 items that is $J \quad I_1 \quad I_2 \quad\quad I_k$ then the cores of GOTO I_1 X GOTO I_2 X GOTO I_k X are the same since I_1 I_2 I_k all have the same core Let K be the union of all sets of items having the same core as GOTO I_1 X Then GOTO J X K

□

The table produced by Algorithm 4 59 is called the *LALR parsing table* for G If there are no parsing action con icts then the given grammar is said to be an *LALR 1 grammar* The collection of sets of items constructed in step 3 is called the *LALR 1 collection*

Example 4 60 Again consider grammar 4 55 whose GOTO graph was shown in Fig 4 41 As we mentioned there are three pairs of sets of items that can be merged I_3 and I_6 are replaced by their union

$$\begin{array}{lll} I_{36} & C & c\,C \;\; c\; d \\ & C & cC \;\; c\; d \\ & C & d \;\; c\; d \end{array}$$

I_4 and I_7 are replaced by their union

$$I_{47} \quad C \quad d \;\; c\; d$$

and I_8 and I_9 are replaced by their union

$$I_{89} \quad C \quad cC \;\; c\; d$$

The LALR action and goto functions for the condensed sets of items are shown in Fig 4 43

STATE	ACTION			GOTO	
	c	d		S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Figure 4 43 LALR parsing table for the grammar of Example 4 54

To see how the GOTO s are computed consider GOTO I_{36} C In the original set of LR 1 items GOTO I_3 C I_8 and I_8 is now part of I_{89} so we make GOTO I_{36} C be I_{89} We could have arrived at the same conclusion if we considered I_6 the other part of I_{36} That is GOTO I_6 C I_9 and I_9 is now part of I_{89} For another example consider GOTO I_2 c an entry that is exercised after the shift action of I_2 on input c In the original sets of LR 1 items GOTO I_2 c I_6 Since I_6 is now part of I_{36} GOTO I_2 c becomes I_{36} Thus the entry in Fig 4 43 for state 2 and input c is made s36 meaning shift and push state 36 onto the stack □

When presented with a string from the language **c dc d** both the LR parser of Fig 4 42 and the LALR parser of Fig 4 43 make exactly the same sequence of shifts and reductions although the names of the states on the stack may di er For instance if the LR parser puts I_3 or I_6 on the stack the LALR parser will put I_{36} on the stack This relationship holds in general for an LALR grammar The LR and LALR parsers will mimic one another on correct inputs

When presented with erroneous input the LALR parser may proceed to do some reductions after the LR parser has declared an error However the LALR parser will never shift another symbol after the LR parser declares an error For example on input *ccd* followed by the LR parser of Fig 4 42 will put

0 3 3 4

on the stack and in state 4 will discover an error because is the next input symbol and state 4 has action error on In contrast the LALR parser of Fig 4 43 will make the corresponding moves putting

0 36 36 47

on the stack But state 47 on input has action reduce C d The LALR parser will thus change its stack to

0 36 36 89

Now the action of state 89 on input is reduce C cC The stack becomes

0 36 89

whereupon a similar reduction is called for obtaining stack

0 2

Finally state 2 has action error on input so the error is now discovered

4 7 5 E cient Construction of LALR Parsing Tables

There are several modi cations we can make to Algorithm 4 59 to avoid con structing the full collection of sets of LR 1 items in the process of creating an LALR 1 parsing table

First we can represent any set of LR 0 or LR 1 items I by its kernel that is by those items that are either the initial item S' S or S' S or that have the dot somewhere other than at the beginning of the production body

We can construct the LALR 1 item kernels from the LR 0 item kernels by a process of propagation and spontaneous generation of lookaheads that we shall describe shortly

If we have the LALR 1 kernels we can generate the LALR 1 parsing table by closing each kernel using the function CLOSURE of Fig 4 40 and then computing table entries by Algorithm 4 56 as if the LALR 1 sets of items were canonical LR 1 sets of items

Example 4 61 We shall use as an example of the e cient LALR 1 table construction method the non SLR grammar from Example 4 48 which we re produce below in its augmented form

S'	S
S	L R | R
L	R | **id**
R	L

The complete sets of LR 0 items for this grammar were shown in Fig 4 39 The kernels of these items are shown in Fig 4 44 □

I_0	S' S	I_5	L **id**
I_1	S' S	I_6	S L R
I_2	S L R R L	I_7	L R
I_3	S R	I_8	R L
I_4	L R	I_9	S L R

Figure 4 44 Kernels of the sets of LR 0 items for grammar 4 49

Now we must attach the proper lookaheads to the LR 0 items in the kernels to create the kernels of the sets of LALR 1 items There are two ways a lookahead b can get attached to an LR 0 item B in some set of LALR 1 items J

1 There is a set of items I with a kernel item A a and J GOTO I X and the construction of

$$\text{GOTO} \;\; \text{CLOSURE} \; \{ A \qquad a \} \;\; X$$

as given in Fig 4 40 contains B b regardless of a Such a looka head b is said to be generated *spontaneously* for B As a special case lookahead is generated spontaneously for the item S' S in the initial set of items

2 All is as in 1 but a b and GOTO CLOSURE $\{ A$ $b \}$ X as given in Fig 4 40 contains B b only because A has b as one of its associated lookaheads In such a case we say that lookaheads *propagate* from A in the kernel of I to B in the kernel of J Note that propagation does not depend on the particular lookahead symbol either all lookaheads propagate from one item to another or none do

We need to determine the spontaneously generated lookaheads for each set of LR 0 items and also to determine which items propagate lookaheads from which The test is actually quite simple Let be a symbol not in the grammar at hand Let A be a kernel LR 0 item in set I Compute for each X J GOTO CLOSURE $\{ A \}$ X For each kernel item in J we examine its set of lookaheads If is a lookahead then lookaheads propagate to that item from A Any other lookahead is spontaneously generated These ideas are made precise in the following algorithm which also makes use of the fact that the only kernel items in J must have X immediately to the left of the dot that is they must be of the form B X

Algorithm 4 62 Determining lookaheads

INPUT The kernel K of a set of LR 0 items I and a grammar symbol X

OUTPUT The lookaheads spontaneously generated by items in I for kernel items in GOTO I X and the items in I from which lookaheads are propagated to kernel items in GOTO I X

METHOD The algorithm is given in Fig 4 45 □

```
for  each item $A$        in $K$  {
       $J$   CLOSURE $\{ A$          $\}$
       if  $B$      $X$   $a$  is in $J$  and $a$ is not
              conclude that lookahead $a$ is generated spontaneously for item
                   $B$      $X$   in GOTO $I$ $X$
       if  $B$      $X$       is in $J$
              conclude that lookaheads propagate from $A$         in $I$ to
                   $B$      $X$   in GOTO $I$ $X$
}
```

Figure 4 45 Discovering propagated and spontaneous lookaheads

We are now ready to attach lookaheads to the kernels of the sets of LR 0 items to form the sets of LALR 1 items First we know that is a looka head for S' S in the initial set of LR 0 items Algorithm 4 62 gives us all the lookaheads generated spontaneously After listing all those lookaheads we must allow them to propagate until no further propagation is possible There are many di erent approaches all of which in some sense keep track of new lookaheads that have propagated into an item but which have not yet propa gated out The next algorithm describes one technique to propagate lookaheads to all items

Algorithm 4 63 E cient computation of the kernels of the LALR 1 collec tion of sets of items

INPUT An augmented grammar G'

OUTPUT The kernels of the LALR 1 collection of sets of items for G'

METHOD

1 Construct the kernels of the sets of LR 0 items for G If space is not at a premium the simplest way is to construct the LR 0 sets of items as in Section 4 6 2 and then remove the nonkernel items If space is severely constrained we may wish instead to store only the kernel items for each set and compute GOTO for a set of items I by rst computing the closure of I

2 Apply Algorithm 4 62 to the kernel of each set of LR 0 items and gram mar symbol X to determine which lookaheads are spontaneously gener ated for kernel items in GOTO I X and from which items in I lookaheads are propagated to kernel items in GOTO I X

3 Initialize a table that gives for each kernel item in each set of items the associated lookaheads Initially each item has associated with it only those lookaheads that we determined in step 2 were generated sponta neously

4 Make repeated passes over the kernel items in all sets When we visit an item i we look up the kernel items to which i propagates its lookaheads using information tabulated in step 2 The current set of lookaheads for i is added to those already associated with each of the items to which i propagates its lookaheads We continue making passes over the kernel items until no more new lookaheads are propagated

□

Example 4 64 Let us construct the kernels of the LALR 1 items for the grammar of Example 4 61 The kernels of the LR 0 items were shown in Fig 4 44 When we apply Algorithm 4 62 to the kernel of set of items I_0 we rst compute CLOSURE $\{ S' \quad S \quad \}$ which is

$$\begin{array}{llll} S' & S & L & R \\ S & L \quad R & L & \textbf{id} \\ S & R & R & L \end{array}$$

Among the items in the closure we see two where the lookahead has been generated spontaneously The rst of these is L R This item with to the right of the dot gives rise to L R That is is a spontaneously generated lookahead for L R which is in set of items I_4 Similarly L **id** tells us that is a spontaneously generated lookahead for L **id** in I_5

As is a lookahead for all six items in the closure we determine that the item S' S in I_0 propagates lookaheads to the following six items

$S' \quad S$ in I_1	$L \quad R$ in I_4
$S \quad L \quad R$ in I_2	$L \quad$ **id** in I_5
$S \quad R$ in I_3	$R \quad L$ in I_2

FROM	TO
$I_0 \quad S' \quad S$	$I_1 \quad S' \quad S$
	$I_2 \quad S \quad L \quad R$
	$I_2 \quad R \quad L$
	$I_3 \quad S \quad R$
	$I_4 \quad L \quad R$
	$I_5 \quad L \quad$ **id**
$I_2 \quad S \quad L \quad R$	$I_6 \quad S \quad L \quad R$
$I_4 \quad L \quad R$	$I_4 \quad L \quad R$
	$I_5 \quad L \quad$ **id**
	$I_7 \quad L \quad R$
	$I_8 \quad R \quad L$
$I_6 \quad S \quad L \quad R$	$I_4 \quad L \quad R$
	$I_5 \quad L \quad$ **id**
	$I_8 \quad R \quad L$
	$I_9 \quad S \quad L \quad R$

Figure 4 46   Propagation of lookaheads

In Fig 4 47 we show steps 3 and 4 of Algorithm 4 63 The column labeled INIT shows the spontaneously generated lookaheads for each kernel item These are only the two occurrences of discussed earlier and the spontaneous lookahead for the initial item $S' \quad S$

On the rst pass the lookahead propagates from $S' \quad S$ in I_0 to the six items listed in Fig 4 46 The lookahead propagates from $L \quad R$ in I_4 to items $L \quad R$ in I_7 and $R \quad L$ in I_8 It also propagates to itself and to L **id** in I_5 but these lookaheads are already present In the second and third passes the only new lookahead propagated is discovered for the successors of I_2 and I_4 on pass 2 and for the successor of I_6 on pass 3 No new lookaheads are propagated on pass 4 so the nal set of lookaheads is shown in the rightmost column of Fig 4 47

Note that the shift reduce con ict found in Example 4 48 using the SLR method has disappeared with the LALR technique The reason is that only lookahead is associated with $R \quad L$ in I_2 so there is no con ict with the parsing action of shift on generated by item $S \quad L \quad R$ in I_2 □

SET	ITEM	LOOKAHEADS			
		INIT	PASS 1	PASS 2	PASS 3
I_0	$S' \quad S$				
I_1	$S' \quad S$				
I_2	$S \quad L \quad R$ $R \quad L$				
I_3	$S \quad R$				
I_4	$L \quad R$				
I_5	L **id**				
I_6	$S \quad L \quad R$				
I_7	$L \quad R$				
I_8	$R \quad L$				
I_9	$S \quad L \quad R$				

Figure 4 47 Computation of lookaheads

4 7 6 Compaction of LR Parsing Tables

A typical programming language grammar with 50 to 100 terminals and 100 productions may have an LALR parsing table with several hundred states The action function may easily have 20 000 entries each requiring at least 8 bits to encode On small devices a more e cient encoding than a two dimensional array may be important We shall mention brie y a few techniques that have been used to compress the ACTION and GOTO elds of an LR parsing table

One useful technique for compacting the action eld is to recognize that usually many rows of the action table are identical For example in Fig 4 42 states 0 and 3 have identical action entries and so do 2 and 6 We can therefore save considerable space at little cost in time if we create a pointer for each state into a one dimensional array Pointers for states with the same actions point to the same location To access information from this array we assign each terminal a number from zero to one less than the number of terminals and we use this integer as an o set from the pointer value for each state In a given state the parsing action for the ith terminal will be found i locations past the pointer value for that state

Further space e ciency can be achieved at the expense of a somewhat slower parser by creating a list for the actions of each state The list consists of terminal symbol action pairs The most frequent action for a state can be placed at the end of the list and in place of a terminal we may use the notation **any** meaning that if the current input symbol has not been found so far on the list we should do that action no matter what the input is Moreover error entries can safely be replaced by reduce actions for further uniformity along a row The errors will be detected later before a shift move

Example 4 65 Consider the parsing table of Fig 4 37 First note that the actions for states 0 4 6 and 7 agree We can represent them all by the list

SYMBOL	ACTION
id	s5
	s4
any	error

State 1 has a similar list

	s6
	acc
any	error

In state 2 we can replace the error entries by r2 so reduction by production 2 will occur on any input but Thus the list for state 2 is

	s7
any	r2

State 3 has only error and r4 entries We can replace the former by the latter so the list for state 3 consists of only the pair **any** r4 States 5 10 and 11 can be treated similarly The list for state 8 is

	s6
	s11
any	error

and for state 9

	s7
any	r1

□

We can also encode the GOTO table by a list but here it appears more e cient to make a list of pairs for each nonterminal A Each pair on the list for A is of the form *currentState nextState* indicating

$$\text{GOTO } currentState \;\; A \qquad nextState$$

This technique is useful because there tend to be rather few states in any one column of the GOTO table The reason is that the GOTO on nonterminal A can only be a state derivable from a set of items in which some items have A immediately to the left of a dot No set has items with X and Y immediately to the left of a dot if $X \,/\, Y$ Thus each state appears in at most one GOTO column

For more space reduction we note that the error entries in the goto table are never consulted We can therefore replace each error entry by the most common non error entry in its column This entry becomes the default it is represented in the list for each column by one pair with **any** in place of *currentState*

Example 4 66 Consider Fig 4 37 again The column for F has entry 10 for state 7 and all other entries are either 3 or error We may replace error by 3 and create for column F the list

CURRENTSTATE	NEXTSTATE
7	10
any	3

Similarly a suitable list for column T is

6	9
any	2

For column E we may choose either 1 or 8 to be the default two entries are necessary in either case For example we might create for column E the list

4	8
any	1

□

This space savings in these small examples may be misleading because the total number of entries in the lists created in this example and the previous one together with the pointers from states to action lists and from nonterminals to next state lists result in unimpressive space savings over the matrix imple mentation of Fig 4 37 For practical grammars the space needed for the list representation is typically less than ten percent of that needed for the matrix representation The table compression methods for nite automata that were discussed in Section 3 9 8 can also be used to represent LR parsing tables

4 7 7 Exercises for Section 4 7

Exercise 4 7 1 Construct the

a canonical LR and

b LALR

sets of items for the grammar $S \quad S\ S \quad |\ S\ S \quad |\ a$ of Exercise 4 2 1

Exercise 4 7 2 Repeat Exercise 4 7 1 for each of the augmented grammars of Exercise 4 2 2 a g

Exercise 4 7 3 For the grammar of Exercise 4 7 1 use Algorithm 4 63 to compute the collection of LALR sets of items from the kernels of the LR 0 sets of items

Exercise 4 7 4 Show that the following grammar

$$\begin{array}{ll} S & A\ a \mid b\ A\ c \mid d\ c \mid b\ d\ a \\ A & d \end{array}$$

is LALR 1 but not SLR 1

Exercise 4 7 5 Show that the following grammar

$$\begin{array}{ll} S & A\ a \mid b\ A\ c \mid B\ c \mid b\ B\ a \\ A & d \\ B & d \end{array}$$

is LR 1 but not LALR 1

4 8 Using Ambiguous Grammars

It is a fact that every ambiguous grammar fails to be LR and thus is not in any of the classes of grammars discussed in the previous two sections How ever certain types of ambiguous grammars are quite useful in the speci cation and implementation of languages For language constructs like expressions an ambiguous grammar provides a shorter more natural speci cation than any equivalent unambiguous grammar Another use of ambiguous grammars is in isolating commonly occurring syntactic constructs for special case optimiza tion With an ambiguous grammar we can specify the special case constructs by carefully adding new productions to the grammar

Although the grammars we use are ambiguous in all cases we specify dis ambiguating rules that allow only one parse tree for each sentence In this way the overall language speci cation becomes unambiguous and sometimes it be comes possible to design an LR parser that follows the same ambiguity resolving choices We stress that ambiguous constructs should be used sparingly and in a strictly controlled fashion otherwise there can be no guarantee as to what language is recognized by a parser