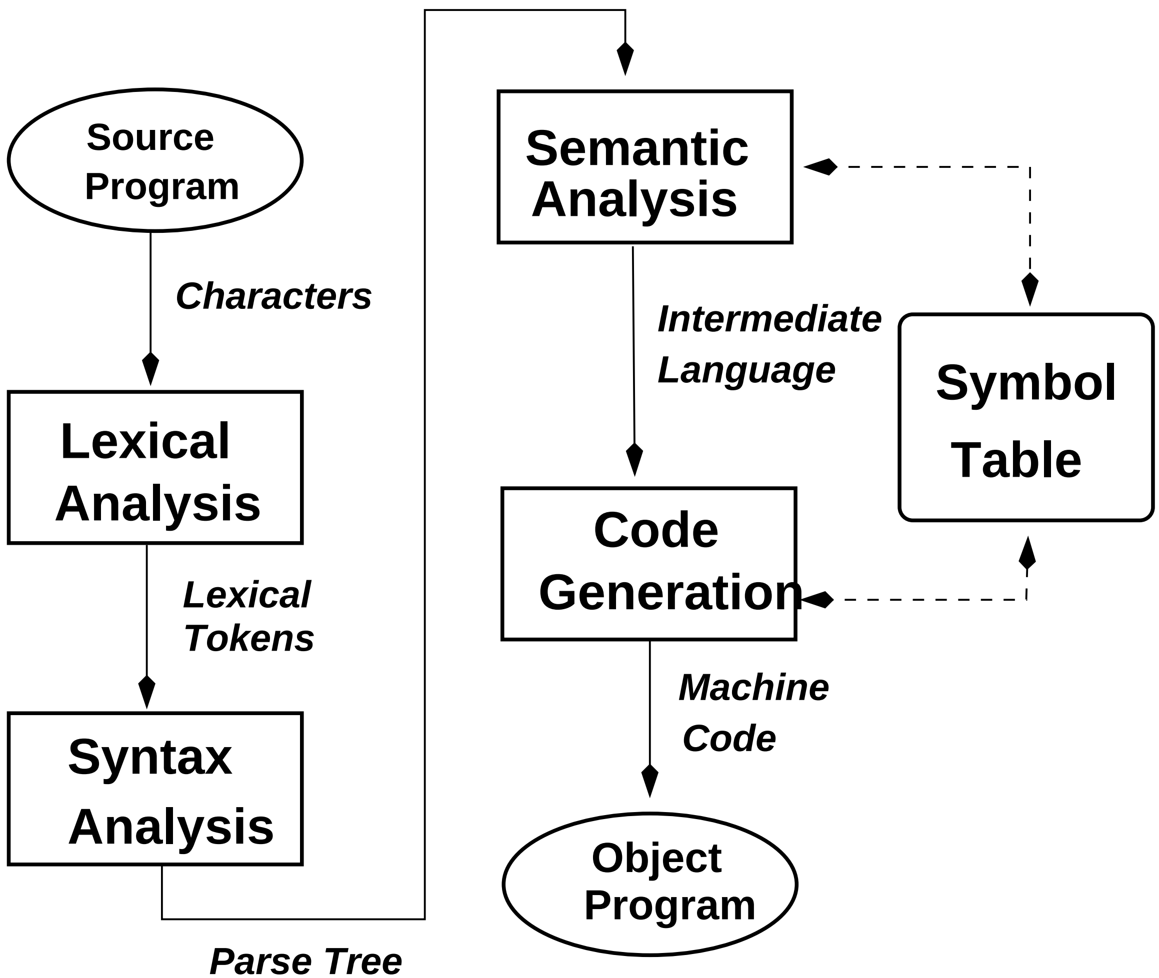
CSC488/2107S Winter 2019

Compilers & Interpeters

`https://www.cs.toronto.edu/~csc488h/`

Week 5 notes

Simple Generic Compiler



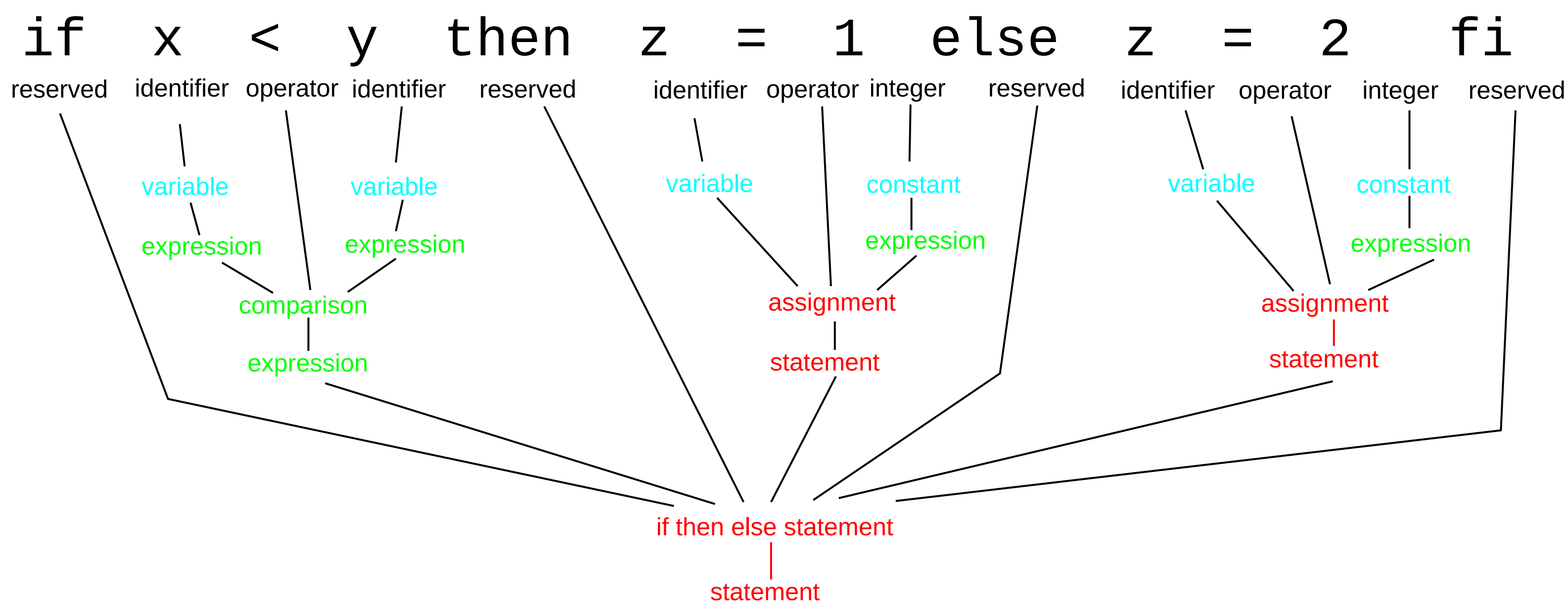
Source statement

```
if  x  < y  then  z  =  1  else  z  =  2   fi
```

Lexical analysis

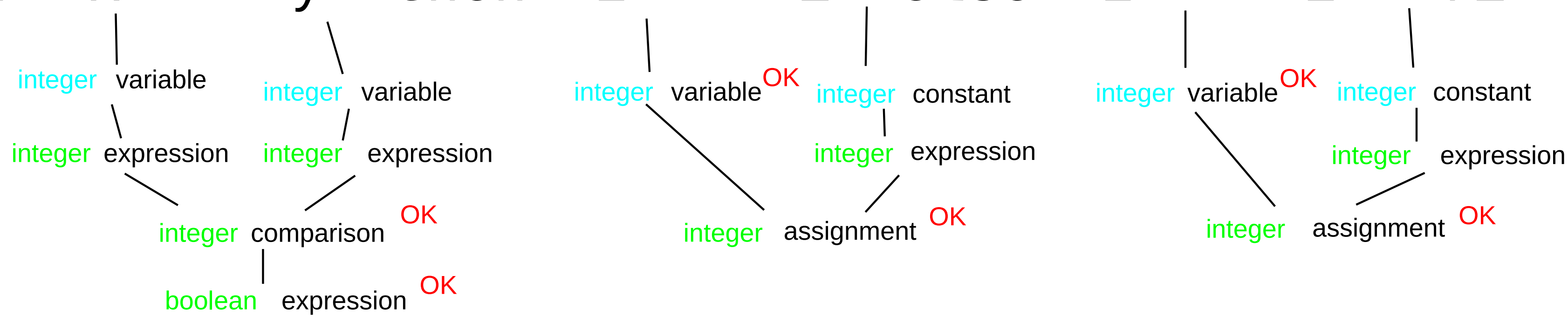
if	x	<	y	then	z	=	1	else	z	=	2	fi
reserved	identifier	operator	identifier	reserved	identifier	operator	integer	reserved	identifier	operator	integer	reserved

Syntax analysis



Semantic analysis

`if x < y then z = 1 else z = 2 fi`



Code Generation

`if x < y then z = 1 else z = 2 fi`

```
load     r1,x
load     r2,y
less     r1,r2
brfalse  L23

load     r1,=1
loadaddr r2,z
store    r2,r1
branch   L24

L23: load     r1,=2
     loadaddr r2,z
     store    r2,r1
L24:
```

Syntax Analysis – Good Language Design

- Use reserved words not keywords
- Design statements and declarations for ease of parsing
  e.g. statements start with distinct reserved words.
- Use sufficient bracketing so endings are clear.

  ambiguous **return** and **return** expression

  unambiguous **return** and **return** (expression)

  Semicolons as statement terminators are good.
- Unambiguous syntax

  ambiguous: **if** expression **then** statement **else** statement

  unambiguous: **if** expression **then** statement **else** statement **fi**
- Design for parsing with one token look ahead.

Syntacticallly Challenged Language - Python

The Python programming language uses *indentation* to mark the beginning and end of blocks. This includes delimiting the bodies of functions and the bodies of control statements.

Python	Description
def calc(x) ;	define function calc
n = x * x + 7	assignment statement in calc
return n * n + 5	return statement in calc
	end of calc
def map (n , m)	define function map
if n < m :	begin body of map
i = n - m	body of if statement
j = n + m	if statement continues
if n > m :	start new if statement
i = n * m + 7	body of if statement
j = i * 2 + 5	if statement continues
return k - 17	end if statement
	end of map
print map(17, 23)	start of main program

Since the scanner is the only part of the compiler that knows about indentation most of the heavy lifting should be done there. One possible solution.

- Define two lexical tokens `<INDENT>` and `<EXDENT>` that are emitted by the scanner whenever the level of indentation changes.
- Define the compilers parsing grammar in terms of these lexical tokens, e.g.
  - body $\rightarrow$ `<INDENT>` statements `<EXDENT>`
  - statement $\rightarrow$ `<INDENT>` statements `<EXDENT>`
- Suppress these invented tokens in any compiler error messages

Shift Reduce Parsing

Bottom Up - LR(k), SLR(k), LALR(k)

- Parser Model

- Parser Actions

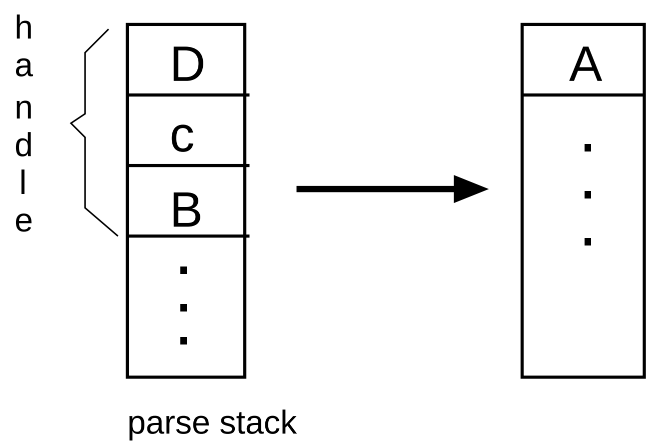
Shift	Next input symbol is pushed onto the stack
Reduce	A sequence of symbols (the *handle*) starting at the top of the stack is reduced using a production rule to replace the symbols with one nonterminal symbol.
Accept	Successful end of parse.
Error	Call recovery routine to handle syntax error.

- Choice of actions is based on the *contents* of the stack (the *left context*) and the next k input tokens (*k-symbol lookahead*).

-

- A *handle* is the right hand side (RHS) of some rule in the grammar. Bottom up parsing allows more than one rule to have the same RHS iff the rules can be distinguished using the left context and k-symbol lookahead.

- Given a grammar rule: A $\rightarrow$ B c D
  a possible Reduce action would be



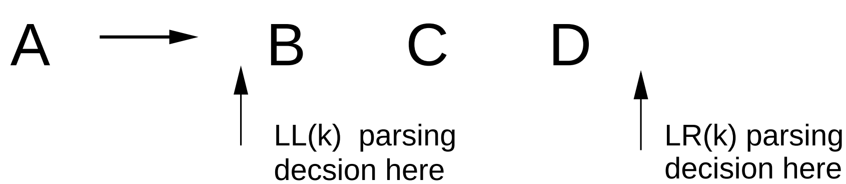
- Issue: efficiently detecting when a handle is present on top of the parse stack.

- Issue: deciding which reduction to perform.

LR(k) Parsing

- The contents of the parser stack (left context) represents a string from which the past input can be derived.
- Inputs are stacked until the top elements in the stack (*the handle*) are a complete alternative (RHS) for some rule.
- When a handle is recognized, a reduction is performed and the handle on the stack is replaced by the nonterminal symbol (LHS) of the applicable rule.
- Initial parser stack is $\triangledown$ and parsing continues until the stack contains $S \triangledown$ and the next input is $\$$
- At each stage the the top elements in the stack represent the initial portion of *one or more* alternative rules.
  The next input symbol may narrow the number of possible alternatives.
  If the number of alternatives is narrowed to zero, a syntax error has occurred.

- Finally, an input symbol is stacked that completes one or more alternatives. If there is more than one alternative, the language is not LR(0).
- At this point the next k input symbols must provide enough information to distinguish among the alternatives. If it doesn't, the language isn't LR(k).
- For LL(k) we had to know at the start of an alternative, given k input symbols which alternative to choose.
  For LR(k) we do not need to know which alternative to choose until we reach the end of a rule. *Then* the next k input symbols must be sufficient to decide if a reduction can be performed.
- Parsing decisions can be made *later* in an LR(k) parser than in an LL(k) parser. This is the reason that $\mathcal{L}(\text{ LL(1) }) \subset \mathcal{L}(\text{ LR(1) })$



LR(k) parsers effectively perform a rightmost derivation in reverse

Rightmost Derivation Example

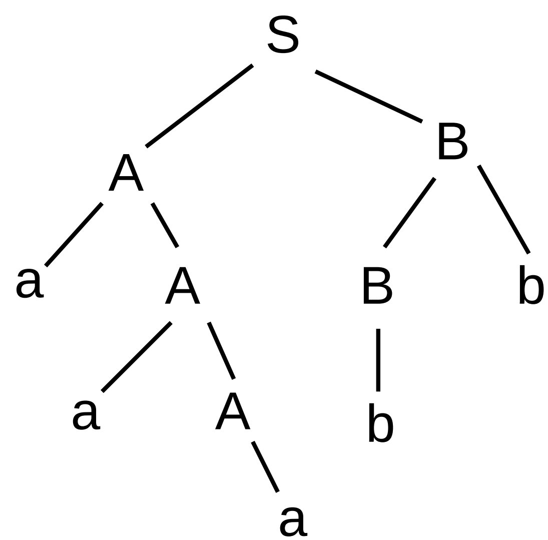
For the grammar:

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

Rightmost derivation of `a a a b b`

0		S
1	$\rightarrow$	A B
2	$\rightarrow$	A B b
3	$\rightarrow$	A b b
4	$\rightarrow$	a A b b
5	$\rightarrow$	a a A b b
6	$\rightarrow$	a a a b b

Parse Tree



Rightmost derivation **in reverse**

6		a a a b b
5	$\rightarrow$	a a A b b
4	$\rightarrow$	a A b b
3	$\rightarrow$	A b b
2	$\rightarrow$	A B b
1	$\rightarrow$	A B
0	$\rightarrow$	S

LR(k) Definition. The symbol $\Rightarrow^*_{rm}$ specifies a *rightmost* derivation

- LR(k) parsers are the most powerful class of deterministic bottom up parsers using k-symbol lookahead.
  If a grammar G can be parsed by any deterministic parser with k-symbol lookahead, then it can be parsed by an LR(k) parser.

- A grammar G is LR(k) if and only if the conditions
  1. $S \Rightarrow^*_{rm} \alpha A \; w \Rightarrow \alpha \beta w$ Identical prefix $\alpha \beta$
  2. $S \Rightarrow^*_{rm} \gamma B \; x \Rightarrow \alpha \beta y$
  3. $First_k(w) = First_k(y)$ Identical lookahead

  imply that $\alpha A y = \gamma B x$

- This means that a reduction $A \rightarrow \beta$ can be performed whenever
  1) $\alpha \beta$ is on top of the parse stack (α is the left context)

  and 2) the k-symbol lookahead is $First_k(w)$.

  $\Rightarrow$ the parser always has enough information to make a parsing decision.

LR(1) Example

Rule	Grammar
1	L $\rightarrow$ L , E
2	$\rightarrow$ E
3	E $\rightarrow$ a
4	$\rightarrow$ b

Input Tokens

a , b $

Note left recursion in grammar $\Rightarrow$ not LL(k)

LR(1) Table

Stack Confg	a	b	,	$	St#
▽	Shift a Next	Shift b Next	Error	Error	0
a ▽	Error	Error	Reduce 3	Reduce 3	1
b ▽	Error	Error	Reduce 4	Reduce 4	2
E ▽	Error	Error	Reduce 2	Reduce 2	3
L ▽	Error	Error	Shift , Next	Accept	4
, L ▽	Shift a Next	Shift b Next	Error	Error	5
a , L ▽	Error	Error	Reduce 3	Reduce 3	6
b , L ▽	Error	Error	Reduce 4	Reduce 4	7
E , L ▽	Error	Error	Reduce 1	Reduce 1	8

Stack Snapshots

Parse	State	Input	Action
▽	▽	a	Next
a ▽	1 ▽	,	
E ▽	3 ▽	,	
L ▽	4 ▽	,	Next
, L ▽	5 4 ▽	b	Next
b , L ▽	7 5 4 ▽	$	
E , L ▽	8 5 4 ▽	$	
L ▽	4 ▽	$	Accept

Define: Reduce j , use grammar rule j to replace handle on top of parse stack.

LR(1) Parse Tables

- Some rows in table are the same as others , e.g. rows 1/6 , 2/7 in the previous slide. To reduce table size these rows can be merged and assigned multiple indices.
- If the grammar is right recursive , the number of different parse stacks is infinite , but the number of different rows is finite (bounded by number of actions $\times$ number of columns) so we *must* merge the rows that are the same.
- Don't want the inefficiency of pattern matching the top elements in the stack against alternative stack configurations.
  We want parsing action to be determined by the single top element on the stack and a single input symbols (as in LL(1)).
- Assign a state number to each row in the table and stack the state number as a synonym for a complete stack configuration.
  The state number is labeled St# in the previous Slide.
  Top state on parse stack represents entire stack configuration

- Redefine: Shift i

  Stack input symbol   Advance input   Go to State i

- After a Reduce , need a list of states to restart in , the *new state table*. With this table we don't need to represent the stack configurations directly in the parser tables.

- Redefine Reduce i

  – Remove handle by doing pop^l where l is the length of the alternative i.

    Need a table giving the length of each alternative.

    This popping uncovers some state. Note state 0 is never popped.

  – Push a new state where new state is a function of top item in the state stack **and** the nonterminal symbol that is being **reduced to** (the LHS of alternative i) State 0 on top signifies the empty stack. For example in the next slide:

Reducing	Top state	Push state	From state	Rule(s)
E	0	3	1 , 2	3 , 4
E	5	6	1 , 2	3 , 4
L	0	4	3	2
L	6	4	6	1

LR(1) Example Revisited

Grammar

1	L	$\rightarrow$	L , E
2		$\rightarrow$	E
3	E	$\rightarrow$	a
4		$\rightarrow$	b

Condensed Parse Table

	Stack	St#	a	b	,	$
	▽	0	Shift 1	Shift 2		
a ▽	a , L ▽	1			Reduce 3	Reduce 3
b ▽	b , L ▽	2			Reduce 4	Reduce 4
	E ▽	3			Reduce 2	Reduce 2
	L ▽	4			Shift 5	Accept
	, L ▽	5	Shift 1	Shift 2		
	E , L ▽	6			Reduce 1	Reduce 1

Example: parse of a , b $

Old Stack	State Stack	Input	Action
▽	0	a	Shift 1
a ▽	1 0	,	Reduce 3
E ▽	3 0	,	Reduce 2
L ▽	4 0	,	Shift 5
, L ▽	5 4 0	b	Shift 2
b , L ▽	2 5 4 0	$	Reduce 4
E , L ▽	6 5 4 0	$	Reduce 1
L ▽	4 0	$	Accept

Theoretical LR(1) Table Construction

- **for all** input symbols ($\Sigma \cup \{ \$ \}$) /* table columns */

  for all possible stack configurations /* table rows */

  /* Fill one parse table entry */

  Compute action(parseStackConfiguration , inputSymbol)

  /* Compute one next state table entry */

  Compute nextState(parseStackConfiguration , inputSymbol)

- There are a finite number of possible stack configurations for a finite grammar.
- Use *parser state numbers* to encode stack configurations so decisions can be made based on state number instead of a pattern match on the stack.
- LR(1) tables for real grammars are very large due to the large number of possible stack configurations

  e.g. > 1240 states and $> 10,000$ table entries for for Pascal.
- For LR(k) iterate the columns over $\Sigma^k \cup \{ \$ \}^k$

Practical LR(1) Parse Table Construction

1. First Compute LR(0) tables:
   (a) LR(0) uses no lookahead
   (b) Apply Closure and Completion to enumerate all possible stack configurations
   (c) Note Conflicts in tables when lookahead is needed.
   Rules for which grammar is not LL(0)
2. Upgrade From LR(0) to LR(1)
   - Use exact lookahead to resolve LR(0) table conflicts
   - Split states as required to force unique left context and lookahead for every conflicting rule.
3. SLR – **S**imple LR , uses Follow sets instead of lookahead to resolve LR(0) conflicts.
4. LALR – **L**ook**A**head LR , use *Item lookahead* to make specific parsing decisions.

LR(0) Table Construction

- Each *parser state* in an LR parser is associated with a unique **item set** of LR(0) items (partially completed phrases).

  The LR(0) item represents what has been seen *prior* to entering the state.

- Define: LR(0) item

  A LR(0) item is a marked production rule

  $$G \rightarrow \alpha \bullet \beta \gamma$$

  with $\alpha, \gamma \;\; in \;\; (N \cup \Sigma)^*$ and $\beta \;\; in \;\; (N \cup \Sigma)$

  An LR(0) item represents a partial phrase ,

  α seen so far $\beta\,\gamma$ to be seen

  β is a single terminal or nonterminal symbol

- An LR(0) item with the bookmark $\bullet$ at the right end. e.g.

  $$G \rightarrow \alpha \beta \gamma \bullet$$

  is a *REDUCE* production , $\alpha \beta \gamma$ is the handle , reduce to G

- To generate an LR(0) table , start with an item set that contains all of the productions with a marker at the start of the right hand side of each rule. This is the start state of the parser (nothing has been see yet).
- Generate additional item sets (parser states) by applying closure and completion until all item sets have been generated.
  Derive parser state transitions from the item sets.
- Define: closure

  if $\bullet$ is immediately to the left of a nonterminal symbol B

  Add to the item set all *new* LR(0) items such that

  B is the left hand side of a rule , i.e. $B \rightarrow \bullet\ \omega$
- Define: completion

  Collect together in a new item set all LR(0) items

  that have the same symbol after the $\bullet$ (e.g. β)

  Complete by moving the $\bullet$ past the symbol , e.g. $G \rightarrow \alpha\, \beta \bullet \gamma$

  if β is a terminal symbol then this corresponds to *SHIFT* β
- Save space by eliminating duplicate configuration sets as they are generated.

LR(0) Table Construction Example

Grammar:

1:	L	$\rightarrow$	L , E
2:		$\rightarrow$	E
3:	E	$\rightarrow$	a
4:		$\rightarrow$	b

- Augment the set of productions with the rule

  $$0 : \text{Accept} \rightarrow \text{L} \ \$$$

- Item sets:

  $$0 : \{ \text{Accept} \rightarrow \bullet \text{L} \ \$ \quad \text{L} \rightarrow \bullet \text{L} \ , \text{E} \quad \text{L} \rightarrow \bullet \text{E} \quad \text{E} \rightarrow \bullet \text{a} \quad \text{E} \rightarrow \bullet \text{b} \}$$

LR(0) Table Construction Example

Grammar:

1:	L	$\rightarrow$	L , E
2:		$\rightarrow$	E
3:	E	$\rightarrow$	a
4:		$\rightarrow$	b

- Augment the set of productions with the rule

  0 : Accept $\rightarrow$ L \$

- Item sets:

  0 : { Accept $\rightarrow$ • L \$ L $\rightarrow$ • L , E   L $\rightarrow$ • E   E $\rightarrow$ • a   E $\rightarrow$ • b }

  1 : { Accept $\rightarrow$ L • \$ L $\rightarrow$ L • , E }

LR(0) Table Construction Example

Grammar:

1:	L	$\rightarrow$	L , E
2:		$\rightarrow$	E
3:	E	$\rightarrow$	a
4:		$\rightarrow$	b

- Augment the set of productions with the rule

  $$0 : \text{Accept} \rightarrow \text{L } \$$$

- Item sets:

  $0 : \{ \text{Accept} \rightarrow \bullet \text{ L } \$ \quad \text{L} \rightarrow \bullet \text{ L , E} \quad \text{L} \rightarrow \bullet \text{ E} \quad \text{E} \rightarrow \bullet \text{ a} \quad \text{E} \rightarrow \bullet \text{ b} \}$

  $1 : \{ \text{Accept} \rightarrow \text{L} \bullet \$ \quad \text{L} \rightarrow \text{L} \bullet \text{ , E} \}$

  $2 : \{ \text{L} \rightarrow \text{E} \bullet \}$

LR(0) Table Construction Example

Grammar:

1:	L	$\rightarrow$	L , E
2:		$\rightarrow$	E
3:	E	$\rightarrow$	a
4:		$\rightarrow$	b

- Augment the set of productions with the rule

  $$0 : \text{Accept} \rightarrow \text{L } \$$$

- Item sets:

  $0 : \{ \text{Accept} \rightarrow \bullet \text{ L } \$ \quad \text{L} \rightarrow \bullet \text{ L , E} \quad \text{L} \rightarrow \bullet \text{ E} \quad \text{E} \rightarrow \bullet \text{ a} \quad \text{E} \rightarrow \bullet \text{ b} \}$

  $1 : \{ \text{Accept} \rightarrow \text{L} \bullet \$ \quad \text{L} \rightarrow \text{L} \bullet \text{, E} \}$

  $2 : \{ \text{L} \rightarrow \text{E} \bullet \}$

  $3 : \{ \text{E} \rightarrow \text{a} \bullet \}$

LR(0) Table Construction Example

Grammar:

1:	L	$\rightarrow$	L , E
2:		$\rightarrow$	E
3:	E	$\rightarrow$	a
4:		$\rightarrow$	b

- Augment the set of productions with the rule

  0 : Accept $\rightarrow$ L $\$$

- Item sets:

  0 : { Accept $\rightarrow \bullet$ L $\$$ L $\rightarrow \bullet$ L , E L $\rightarrow \bullet$ E E $\rightarrow \bullet$ a E $\rightarrow \bullet$ b }

  1 : { Accept $\rightarrow$ L $\bullet$ $\$$ L $\rightarrow$ L $\bullet$, E }

  2 : { L $\rightarrow$ E $\bullet$ }

  3 : { E $\rightarrow$ a $\bullet$ }

  4 : { E $\rightarrow$ b $\bullet$ }

LR(0) Table Construction Example

Grammar:

1:	L	$\rightarrow$	L , E
2:		$\rightarrow$	E
3:	E	$\rightarrow$	a
4:		$\rightarrow$	b

- Augment the set of productions with the rule

  $$0 : \text{Accept} \rightarrow \text{L} \; \$$$

- Item sets:

  $0 : \{ \text{Accept} \rightarrow \bullet \text{L} \; \$ \quad \text{L} \rightarrow \bullet \text{L} \; , \text{E} \quad \text{L} \rightarrow \bullet \text{E} \quad \text{E} \rightarrow \bullet \text{a} \quad \text{E} \rightarrow \bullet \text{b} \}$

  $1 : \{ \text{Accept} \rightarrow \text{L} \bullet \; \$ \quad \text{L} \rightarrow \text{L} \bullet \; , \text{E} \}$

  $2 : \{ \text{L} \rightarrow \text{E} \bullet \}$

  $3 : \{ \text{E} \rightarrow \text{a} \bullet \}$

  $4 : \{ \text{E} \rightarrow \text{b} \bullet \}$

  $5 : \{ \text{L} \rightarrow \text{L} \; , \bullet \text{E} \quad \text{E} \rightarrow \bullet \text{a} \quad \text{E} \rightarrow \bullet \text{b} \}$

LR(0) Table Construction Example

Grammar:

$$\begin{array}{llll} 1: & L & \rightarrow & L\ ,\ E \\ 2: & & \rightarrow & E \\ 3: & E & \rightarrow & a \\ 4: & & \rightarrow & b \end{array}$$

- Augment the set of productions with the rule

$$0 : \text{Accept} \rightarrow L\ \$$$

- Item sets:

$0 : \{\ \text{Accept} \rightarrow \bullet\ L\ \$ \quad L \rightarrow \bullet\ L\ ,\ E \quad L \rightarrow \bullet\ E \quad E \rightarrow \bullet\ a \quad E \rightarrow \bullet\ b\ \}$

$1 : \{\ \text{Accept} \rightarrow L\ \bullet\ \$ \quad L \rightarrow L\ \bullet\ ,\ E\ \}$

$2 : \{\ L \rightarrow E\ \bullet\ \}$

$3 : \{\ E \rightarrow a\ \bullet\ \}$

$4 : \{\ E \rightarrow b\ \bullet\ \}$

$5 : \{\ L \rightarrow L\ ,\ \bullet\ E \quad E \rightarrow \bullet\ a \quad E \rightarrow \bullet\ b\ \}$

$6 : \{\ L \rightarrow L\ ,\ E\ \bullet\ \}$

LR(0) Table Construction Example

Grammar:

$$
\begin{array}{llll}
1: & L & \rightarrow & L \, , \, E \\
2: & & \rightarrow & E \\
3: & E & \rightarrow & a \\
4: & & \rightarrow & b
\end{array}
$$

- Augment the set of productions with the rule

$$0 : \text{Accept} \rightarrow L \ \$$$

- Item sets:

$$0 : \{ \ \text{Accept} \rightarrow \bullet \, L \ \$ \qquad L \rightarrow \bullet \, L \, , E \qquad L \rightarrow \bullet \, E \qquad E \rightarrow \bullet \, a \qquad E \rightarrow \bullet \, b \ \}$$

$$1 : \{ \ \text{Accept} \rightarrow L \bullet \ \$ \qquad L \rightarrow L \bullet \, , E \ \}$$

$$2 : \{ \ L \rightarrow E \bullet \ \}$$

$$3 : \{ \ E \rightarrow a \bullet \ \}$$

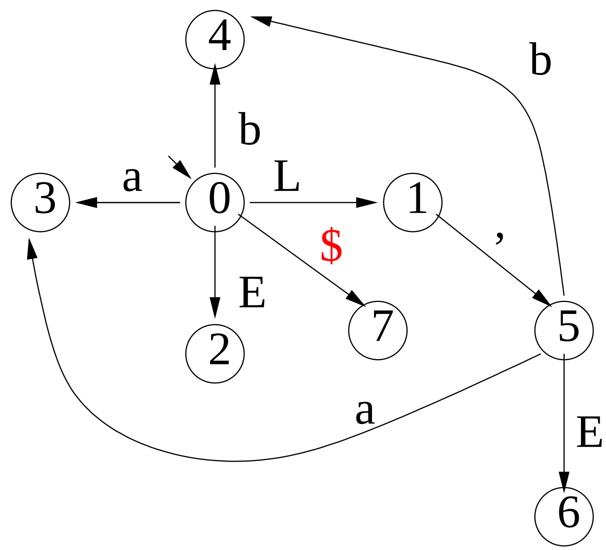
$$4 : \{ \ E \rightarrow b \bullet \ \}$$

$$5 : \{ \ L \rightarrow L \, , \bullet \, E \qquad E \rightarrow \bullet \, a \qquad E \rightarrow \bullet \, b \ \}$$

$$6 : \{ \ L \rightarrow L \, , E \bullet \ \}$$

$$7 : \{ \ \text{Accept} \rightarrow L \ \$ \bullet \ \}$$

LR(0): State Machine view



Grammar

0	Accept	$\rightarrow$	E $
1	L	$\rightarrow$	L , E
2		$\rightarrow$	E
3	E	$\rightarrow$	a
4		$\rightarrow$	b

Parse table

St#	Action
0	Shift
1	Shift
2	Reduce 2
3	Reduce 3
4	Reduce 4
5	Shift
6	Reduce 1
7	Accept

LR(0) Parse Table Construction

- s - set of LR(0) items (table row)
- Building parse table $P(s)$:
  - $\{$If $B \rightarrow \rho \bullet \in s$ and $B \rightarrow \rho$ is numbered i
    then Reduce i $\}$
  - $\{$If $A \rightarrow \alpha \bullet a\beta \in s$ for terminal symbol a
    then Shift
    else $\emptyset\}$
- If $\forall s \cdot \mid P(s) \mid = 1$, the grammar is LR(0).

LR(0) Conflict Diagnosis

- The LR(0) parse table construction has a conflict when it trys to assign more than one value to the parse table entry for some state.
  - **shift/reduce conflicts** Two alternatives exist:
    – Shift the incoming terminal symbol onto the stack
    – Reduce the top of the stack using some rule
  - **reduce/reduce conflicts**. The right hand side of two or more rules match the handle on top of the stack.
- Conflicts may arise because the grammar is ambiguous or because the parse table construction method isn't powerful enough.
- LR(0) conflicts are resolved by using some form of lookahead, i.e. using the next k input symbols to resolve the conflict. Usually $k = 1$.
- Lookahead only matters in cases where the • is at the right end of a production.
  Use the lookahead sets to decide which of several productions to apply.
  *Lookahead sets for each **state** and **production** must distinguish productions uniquely.*

- There are several strategies, e.g. SLR(k), LALR(k) and LR(k) for using lookahead to resolve LR(0) conflicts. These strategies differ
  - in how lookahead information is used
  - in the size of the resultant parse tables
  - in the complexity of the table building algorithm
- SLR(k) (Simple LR) uses $Follow_k$ to resolve conflicts.
- LALR(k) (LookAhead LR)

  Build LR(k)

  Merge all conflicting states that differ *only* in their lookahead.

  Lookahead can then be used to make the parsing decision for these states.
- LR(K) Construct LR(0) states, then augment with lookahead.

  May require splitting states to force unique actions for a given lookahead.

  May result in a very large number of states.

Example 2 – not LR(0)

Grammar:

$$\begin{array}{lcl} S & \rightarrow & E\ \$ \\ E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * P \mid P \\ P & \rightarrow & id \mid (E) \end{array}$$

Item sets:

$0 : \{ S \rightarrow \bullet\ E\ \$\ ,\ E \rightarrow \bullet\ E + T\ ,\ E \rightarrow \bullet\ T\ ,\ T \rightarrow \bullet\ T * P\ ,\ T \rightarrow \bullet\ P\ ,\ P \rightarrow \bullet\ id\ ,\ P \rightarrow \bullet\ (E) \}$

$1 : \{ S \rightarrow E\ \bullet\ \$\ ,\ E \rightarrow E\ \bullet\ + T \}$

$2 : \{ S \rightarrow E\ \$\ \bullet \}$

$3 : \{ E \rightarrow E +\ \bullet\ T\ ,\ T \rightarrow \bullet\ T * P\ ,\ T \rightarrow \bullet\ P\ ,\ P \rightarrow \bullet\ id\ ,\ P \rightarrow \bullet\ (E) \}$

$4 : \{ T \rightarrow P\ \bullet \}$

$5 : \{ P \rightarrow id\ \bullet \}$

$6 : \{ P \rightarrow (\ \bullet\ E)\ ,\ E \rightarrow \bullet\ E + T\ ,\ E \rightarrow \bullet\ T\ ,\ T \rightarrow \bullet\ T * P\ ,\ T \rightarrow \bullet\ P\ ,\ P \rightarrow \bullet\ id\ ,\ P \rightarrow \bullet\ (E) \}$

$7 : \{ E \rightarrow T\ \bullet\ ,\ T \rightarrow T\ \bullet\ * P \}$

$8 : \{ T \rightarrow T *\ \bullet\ P\ ,\ P \rightarrow \bullet\ id\ ,\ P \rightarrow \bullet\ (E) \}$

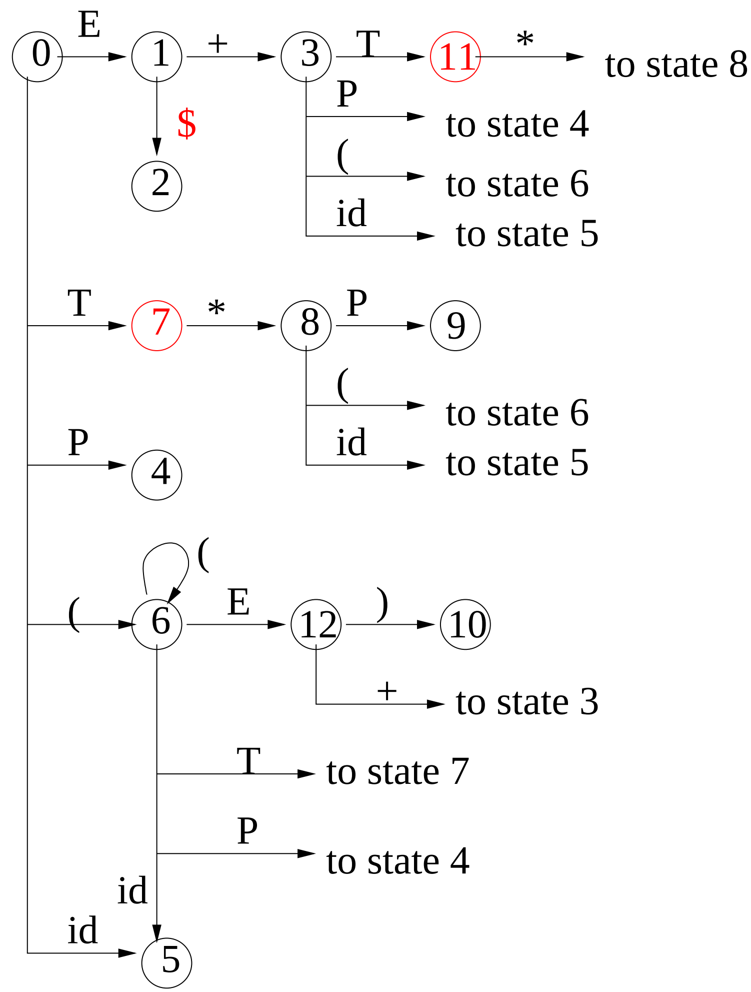
$9 : \{ T \rightarrow T * P\ \bullet \}$

$10 : \{ P \rightarrow (E)\ \bullet \}$

$11 : \{ E \rightarrow E + T\ \bullet\ ,\ T \rightarrow T\ \bullet\ * P \}$

$12 : \{ P \rightarrow (E\ \bullet\ )\ ,\ E \rightarrow E\ \bullet\ + T \}$

LR(0) State Machine for Example 2



- Shift/Reduce conflicts in states 7 and 11 Reduce to E or shift *
- Solution: Have more states (i.e., split states 7 and 11!)

Lookahead and Follow Sets

- Begin with LR(0) but augment with lookahead $L \in \Sigma \cup \{ \$ \} \cup \{\lambda\}$
- Define: *Lookahead Set*

  The lookahead set for a production $A \rightarrow \alpha$ is the set of terminal symbols that can legally follow A **or** α during a rightmost canonical parse.
- Define: *Follow Set*

  For any nonterminal symbol A the set $Follow(A)$ is the set of terminal symbols that can legally follow A in a sentential form during any parse.
- Therefore $lookahead(A) \subset follow(A)$

  Lookahead sets provide more decision making power than Follow sets.

Computing LR(1) Lookahead Sets

- For each marked production the lookahead sets are calculated using closure.
  If $A \rightarrow \alpha \bullet B\,\beta$ $\{$ LookAhead $\}$ is a configuration item and we are adding
  $B \rightarrow \bullet\,\gamma$ $\{$ newLookAhead $\}$ to the configuration set then

  newLookAhead = $first(\beta)$ β is not nullable

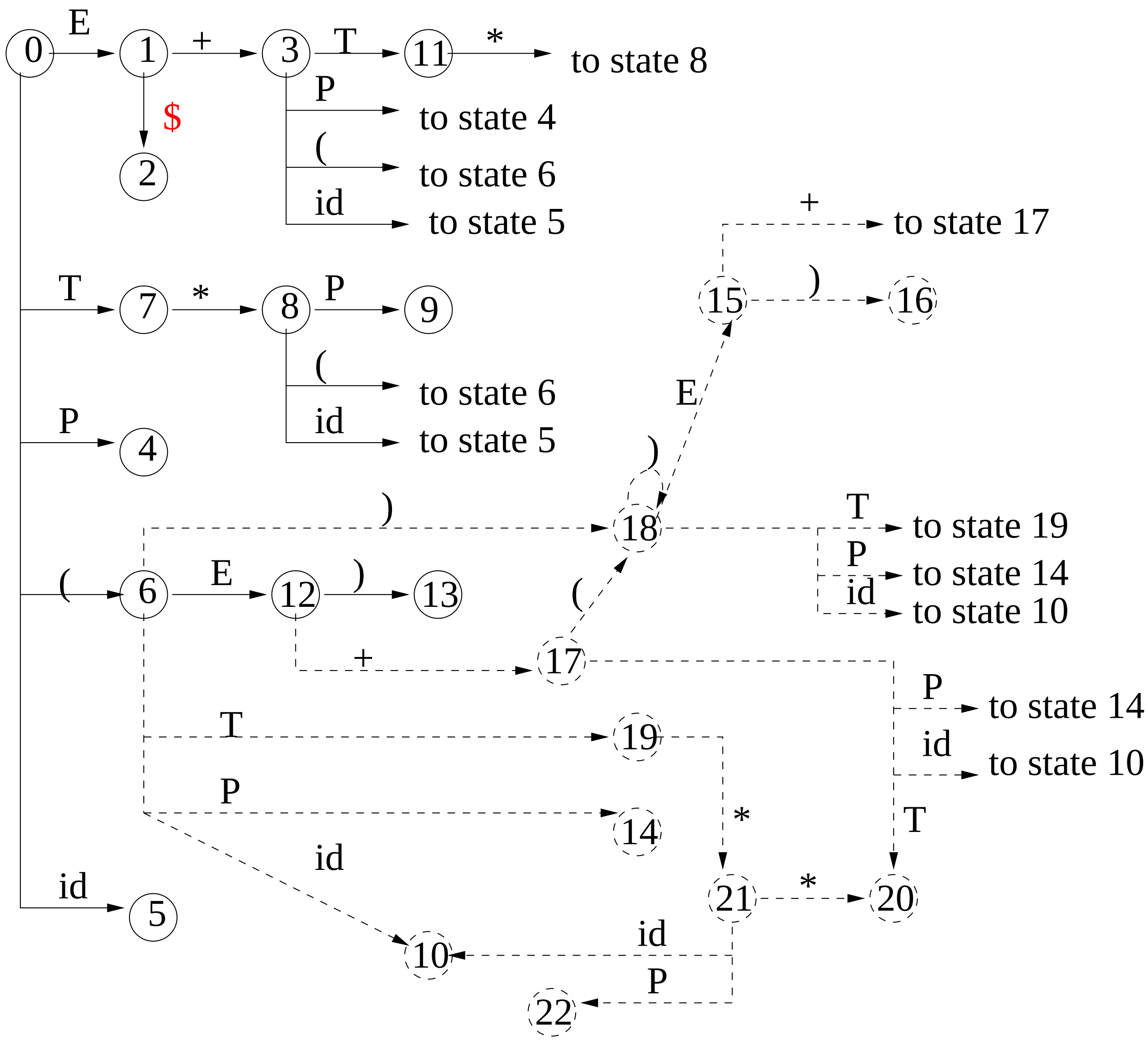
  newLookAhead = $first(\beta) \cup LookAhead$ β is nullable.

- For a marked production obtained by moving a marker past a terminal symbol, the lookahead set is unchanged.

Example

		Lookahead		Lookahead
0	$S \rightarrow \bullet\ E\ \$\,,$	$\{\lambda\}$	$E \rightarrow \bullet\ E + T,$	$\{\$\ +\}$
	$E \rightarrow \bullet\ T,$	$\{\$\ +\}$	$T \rightarrow \bullet\ T * P,$	$\{\$\ +\ *\}$
	$T \rightarrow \bullet\ P,$	$\{\$\ +\ *\}$	$P \rightarrow \bullet\ id,$	$\{\$\ +\ *\}$
	$P \rightarrow \bullet\ (E),$	$\{\$\ +\ *\}$		
7	$E \rightarrow T\ \bullet\ ,$	$\{\$\ +\}$	$T \rightarrow T\ \bullet\ P,$	$\{\$\ +\ *\}$
11	$E \rightarrow E + T\ \bullet\ ,$	$\{\$\ +\}$	$T \rightarrow T\ \bullet\ * P,$	$\{\$\ +\ *\}$
19	$E \rightarrow T\ \bullet\ ,$	$\{)\ +\}$	$T \rightarrow T\ \bullet\ P,$	$\{)\ +\ *\}$
20	$E \rightarrow E + T\ \bullet\ ,$	$\{)\ +\}$	$T \rightarrow T\ \bullet\ * P,$	$\{)\ +\ *\}$

Example LR(1) Statemachine



Building LR(1) Tables

- Building parse table $P(s,a)$:
  - $\{$If $B \rightarrow \rho \bullet \;, \{a\} \in s$ and $B \rightarrow \rho$ is numbered i then Reduce $i\}$
  - $\{$If $A \rightarrow \alpha \bullet a\beta \;, \{a\} \in s$ for terminal a, then Shift else $\emptyset\}$
- If $\forall s, \forall a \cdot \mid P(s,a) \mid = 1$, the grammar is LR(1).

For our example: 23 states instead of 13

State	+	*	id	(	)	\$
0			S	S		
1	S					A
2						
3			S	S		
4	R5	R5				R5
5	R6	R6				R6
6			S	S		
7	R3	S				R3
8			S	S		
9	R4	R4				R4
10	R6	R6			R6	
11	R2	S				R2

State	+	*	id	(	)	\$
12	S				S	
13	R7	R7				R7
14	R5	R5			R5	
15	R7	R7			R7	
16	S				S	
17			S	S		
18			S	S		
19	R3	S			R3	
20	R2	S			R2	
21			S	S		
22	R4	R4			R4	

LALR(1) Parsing

- LALR(1) differs from LR(1) in that states with identical reductions but different lookahead sets are merged in LALR(1) and kept distinct in full LR(1).
  Goal: merge non-essential LR(1) states
- Construct LR(1) table, then merge states (table rows) with the same reductions.
- Define *Cognate*(s'):
  Those states with the same table row as s', union lookaheads.

Example (from LR(1) table in Slide 151):

LALR(1) cognate	0	1	2	3	4	5	6	7	8	9	10	11	12
LR(1) rows	0	1	2	3,17	4,14	5,10	6,18	7,19	8,21	9,22	13,15	11,20	12,16

- Building parse table $P(s,a)$:
  - $\{$If $B \to \rho \bullet, \{a\} \in$ *Cognate*(s) and $B \to \rho$ is numbered i then Reduce $i\}$
  - $\{$If $A \to \alpha \bullet a\beta \in s$ for terminal a, then Shift else $\emptyset\}$
- If $\forall s, a \cdot \mid P(s,a) \mid = 1$, the grammar is LALR(1).

LALR(1) Example (Cont'd)

State 7:

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

State 11:

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

Thus, in both cases, reduce on $\{), \$, +\}$, shift on $*$.

Action table (same as SLR(1))

State	+	*	id	(	)	$
0			S	S		
1	S					A
2						
3			S	S		
4	R5	R5			R5	R5
5	R6	R6			R6	R6
6			S	S		

State	+	*	id	(	)	$
7	R3	S			R3	R3
8			S	S		
9	R4	R4			R4	R4
10	R7	R7			R7	R7
11	R2	S			R2	R2
12	S				S	

Parser Error Recovery

- LL(k) and LR(k) parsers have the valid prefix property:

  if no error has been detected then

  the input thus far is a valid prefix of one or more programs.

  This is a consequence of processing the input token stream strictly left-to right

- Both kinds of parsers can give error messages in terms of input symbols without mentioning the grammar (i.e. names of nonterminal symbols)

- Just look along current row of parse table for non-error entries and list the column headings.

- Changing the parser stack can cause problems in later compiler phases, e.g. compiler internal data structures may be left in an inconsistent state.

  If and only if the parser has the valid prefix property, we can avoid changing the parser stack.

Syntax Error Repair

- Goal: transform syntactically incorrect program into the *closest* correct program, mainly to be able to continue parsing.
  Try to maximize the number of *useful* error messages per compilation.

- Measure of closeness: smallest number of changes. For example:
  insert 1 token or delete 1 token or change 1 token.

- Problems
  - closest correct program may not be unique e.g. delete **begin** or insert **end** ?
  - Algorithms to find a closest correct program are non-linear.
  - Any measure of closeness may not give the intuitively closest program in all cases.
    Examples

PL/I	DOWHILE(I = 1);	$\Rightarrow$	DOWHILE(I) = 1 ;
	Should be	$\Rightarrow$	DO WHILE(I = 1) ;
Turing	gut S	$\Rightarrow$	**get** S
		$\Rightarrow$	**put** S

Syntax Error Repair Strategy

- General strategy: isolate the error in a *replaceable phrase*. Replaceable implies that the parse can continue.
- Making the *closest correction* may require changing the parser stack.
  Example:

  A = B **then** I = 1 **else** I = 0

  The closest correction would be to insert an **if** and compile A = B as a boolean expression. But it may have already been compiled as an assignment statement before we know about the error.
- Three (of many) possible strategies are the *recovery token strategy*, the *panic strategy* and spelling correction.

Syntax Error Repair Strategy

- Recovery token repair strategy
  - For each row in the table, one input token is designated as the *recovery token*.
  - Reserved words, identifiers, numbers and strings are *long* tokens.
    All other tokens are *short* tokens
  - if the incorrect token is long and the recovery token is short
    insert the recovery token in front of the input.
  - Otherwise, replace the current input symbol with the recovery token.
- Panic repair strategy
  - Some tokens are designated as *hard* tokens. For example **;** , **end**
    All other tokens are *soft*
  - Discard input up to and including the first hard token.
  - Pop the parser stack down to a corresponding token or state.
- Spelling correction strategy:
  - Replace identifier that is close to a reserved word with the reserved word.

Parsers for Compilers – Recursive Descent, LL(1) and LR(1)

- Almost all compilers use Recursive Descent, LL(1) or LALR(1)
- LALR(1) is more powerful than LL(1), but LL(1) may be strong enough.
- If you have a parser generator, use it.
- If you have both LL(1) and LALR(1) parser generators, use LALR(1)
- If you don't have a parser generator, use Recursive Descent.
  Or try to manually make the grammar LL(1) and build a simple LL(1) parser
  If you don't have a parser generator *do not* try and build an LALR(1) parser from scratch.
- There are well designed and thoroughly tested open source parser generators for both LL(1) and LALR(1).
- If you're building a scanner/parser for an ugly or complicated language (i.e. C++, Java, Fortran), consider buying an off the shelf compiler front end from a specialist compiler company