CSC 488S/CSC 2107S Lecture Notes

These lecture notes are provided for the personal use of students taking CSC488H1S or CSC2107S in the
Winter 2012/2013 term at the University of Toronto

Reading Assignment

Fischer, Cytron, LeBlanc

Chapter 6

Section 5.9

Shift Reduce Parsing

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

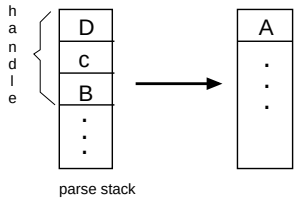
- Parser Model

- 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*).
- See *Fischer, Cytron, LeBlanc* Figure 6.3 for a generic bottom up parsing engine.

- 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

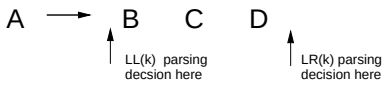


- Issue: efficiently detecting when a handle is present on top of the parse stack.
- Issue: deciding which reduction to perform.
- See *Fischer, Cytron, LeBlanc* Figure 4.6 for an example of a complete bottom up parse.

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 $\bigtriangledown$ and parsing continues until the stack contains $S \bigtriangledown$ and the next input is $\$$
- At each stage the the top elements in the stack represent the initial portion of *one or more* alternatives. The next input symbol narrows the number of number of possible alternatives. If the number of choices in 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 $\mathcal{L}$(LL(k)) $\subset \mathcal{L}$(LR(1))



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

Rightmost Derivation Example[a]

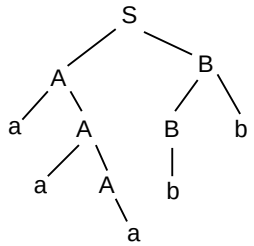
For the grammar:

S → A B
A → a A
  | a
B → B b
  | b

Rightmost derivation of a a a b b

S
→ A B
→ A B b
→ A b b
→ a A b b
→ a a A b b
→ a a a b b

Parse Tree



Rightmost derivation **in reverse**

a a a b b
→ a a A b b
→ a A b b
→ A b b
→ A B b
→ A B
→ S

[a]See Slide 75

LR(k) Definition[a]

- 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
  and 2) the k-symbol lookahead is $First_k(w)$.
  $\Rightarrow$ the parser always has enough information to make a parsing decision.

[a]See *Fischer, Cytron, LeBlanc* Page 190 . The symbol $\Rightarrow_{rm}^{*}$ specifies a *rightmost* derivation

LR(1) Example[a]

Rule	Grammar
1	L → L , E
2	→ E
3	E → a
4	→ b

Input Tokens

a , b $

Note left recursion in grammar ⇒ not LL(k)

LR(1) Table

Stack Confg	a	b	,	$
▽	Shift a	Shift b		
	Next	Next	Error	Error
a ▽	Error	Error	Reduce 3	Reduce 3
b ▽	Error	Error	Reduce 4	Reduce 4
E ▽	Error	Error	Reduce 2	Reduce 2
L ▽			Shift ,	
	Error	Error	Next	Accept
, L ▽	Shift a	Shift b		
	Next	Next	Error	Error
a , L ▽	Error	Error	Reduce 3	Reduce 3
b , L ▽	Error	Error	Reduce 4	Reduce 4
E , L ▽	Error	Error	Reduce 1	Reduce 1

Stack Snapshots

St#	Parse	State	Input	Action
	▽	▽	a	Next
0	a ▽	1 ▽	,	
1	E ▽	3 ▽	,	
2	L ▽	4 ▽	,	Next
3	, L ▽	5 4 ▽	b	Next
	b , L ▽	7 5 4 ▽	$	
4	E , L ▽	8 5 4 ▽	$	
	L ▽	4 ▽	$	Accept
5				
6				
7				
8				

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

[a] Also see the larger example in *Fischer, Cytron, LeBlanc* Figures 6.4, 6.5, 6.6 and 6.7 .

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 × 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	→	L , E
2		→	E
3	E	→	a
4		→	b

Condensed Parse Table

Stack	St#	a	b	,	$
▽	0	Shift 1	Shift 2		
a ▽	1			Reduce 3	Reduce 3
b ▽	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 α, γ *in* $(N \cup \Sigma)^*$ and β *in* $(N \cup \Sigma)$
  An LR(0) item represents a partial phrase ,
  α seen so far $\beta \gamma$ to be seen
- 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 $\beta \varepsilon \Sigma$ then this corresponds to *SHIFT* β
- Save space by eliminating duplicate configuration sets as they are generated.

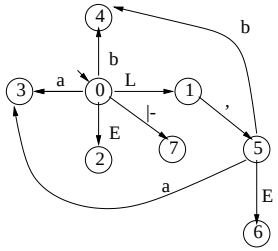
LR(0) Table Construction Example

Grammar:

1: L → L , E
2: → E
3: E → a
4: → b

- Augment the set of productions with the rule
  0 : Accept → L $
- Item sets:
  0 : { Accept → • L $ L → • L , E L → • E E → • a E → • b }
  1 : { Accept → L • $ L → L • , E }
  2 : { L → E • }
  3 : { E → a • }
  4 : { E → b • }
  5 : { L → L , • E E → • a E → • b }
  6 : { L → L , E • }
  7 : { Accept → L $ • }

LR(0): Statemachine view



- 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).

Grammar

0	Accept	→	E $
1	L	→	L , E
2		→	E
3	E	→	a
4		→	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

Another Example – not LR(0)

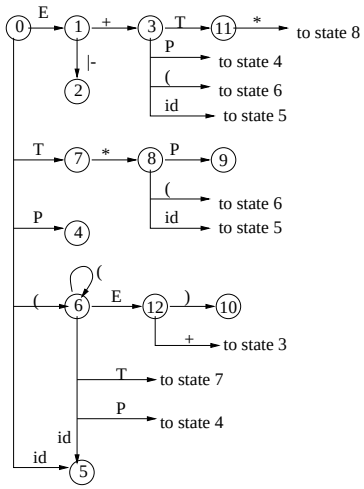
Grammar:

S → E $
E → E + T | T
T → T ∗ P | P
P → id | (E)

Item sets:

0 : { S → • E $, E → • E + T , E → • T , T → • T ∗ P , T → • P , P → • id , P → • (E) }
1 : { S → E • $, E → E • + T }
2 : { S → E $ • }
3 : { E → E + • T , T → • T ∗ P , T → • P , P → • id , P → • (E) }
4 : { T → P • }
5 : { P → id • }
6 : { P → (• E) , E → • E + T , E → • T , T → • T ∗ P , T → • P , P → • id , P → • (E) }
7 : { E → T • , T → T • ∗ P }
8 : { T → T ∗ • P , P → • id , P → • (E) }
9 : { T → T ∗ P • }
10 : { P → (E) • }
11 : { E → E + T • , T → T • ∗ P }
12 : { P → (E •) , E → E • + T }

LR(0) Statemachine for Example 2



- A table conflict occurs when the constructor algorithm tries to assign more than one type of action to some table entry.
- Types of LR(0) conflicts:
  - *Shift/Reduce conflict*, - shift action and reduce action
  - *Reduce/Reduce conflict*, - two different reduce actions
  - Example: Shift/Reduce conflicts in states 7 and 11
    Reduce to E or shift *
- Solution: Have more states (i.e., split states 7 and 11!)

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 $\bullet$ 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) then merge all states that differ in their lookahead. Use the lookahead to make the parsing decision.
- 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.

SLR(1) Parsing

- Goal: get more information than LR(0) but do not create non-essential LR(1) states
- s - set of LR(0) items (table row) , a a terminal symbol (table column)
- Augment CFSM from LR(0) with different notion of lookahead:
- Building $P(s,a)$:
  - $\{$If $B \rightarrow \rho \bullet \in s$, $a \in Follow(B)$ and $B \rightarrow \rho$ is numbered i then
    Reduce i $\}$
  - $\{$If $A \rightarrow \alpha \bullet a\beta \in s$ for terminal a, then
    Shift
    else $\emptyset$ $\}$
- If $\forall s \forall a \cdot \mid P(s,a) \mid = 1$, the grammar is SLR(1).

SLR(1) Example[a]

Follow(S) = $\{\lambda\}$ *Follow*(E) = $\{$ $, +,) $\}$

Follow(P) = $\{$), +, $*$ $\}$ *Follow*(T) = $\{$), +, $*$ $\}$

Follow(E) = $\{$ $, +,)$\}$, so reduce if we see these, shift when we see $*$.

Action table[b]

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

State	Lookahead					
	+	$*$	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	

[a] Grammar from example in Slides 138
[b] **Rn** is **Reduce n** , **S** is **Shift**

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**.
- A terminal symbol b is in $Follow(\ A\ )$ if b can legally follow A in a sentential form during **any** parse. Therefore

  $$lookahead(\ A\ ) \subset follow(\ A\ )$$

  Therefore 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 $A \rightarrow \alpha \bullet B\,\beta$ $\{\ LA\ \}$ is a configuration item and we are adding $B \rightarrow \bullet\,\gamma$ $\{\ newLA\ \}$ to the configuration set then

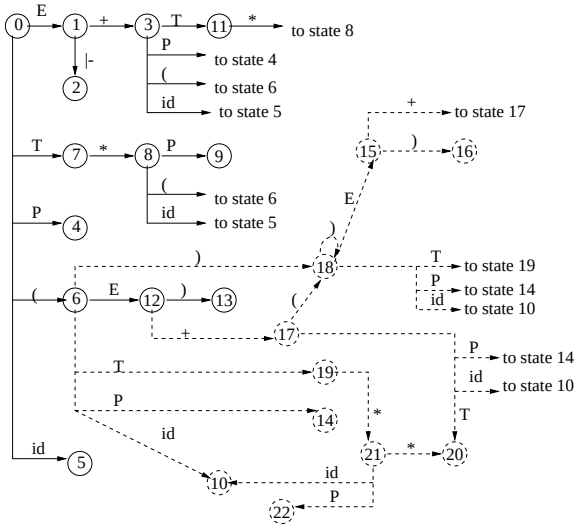
  $newLA = first(\ \beta\ )$ β is not nullable

  $newLA = first(\ \beta\ ) \cup LA$ 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 → • E $,	{ λ }	E → • E + T,	{ $ + }
	E → • T,	{ $ + }	T → • T ∗ P,	{ $ + ∗ }
	T → • P,	{ $ + ∗ }	P → • id,	{ $ + ∗ }
	P → • (E),	{ $ + ∗ }		
7	E → T • ,	{ $ + }	T → T • P,	{ $ + ∗ }
11	E → E + T • ,	{ $ + }	T → T • ∗ P,	{ $ + ∗ }
19	E → T • ,	{) + }	T → T • P,	{) + ∗ }
20	E → E + T • ,	{) + }	T → T • ∗ 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	Lookahead					
	+	∗	id	(	)	$
0			S	S		
1	S					A
2						
3			S	S		
4	R5	R5				R5
5	R6	R6				R6
6			S	S		
7	R3	S				R3
8			S	S		
9	R4	R4				R4
10	R6	R6			R6	
11	R2	S				R2

State	Lookahead					
	+	∗	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 147):

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 $B \rightarrow \rho \bullet$, $\{a\} \in$ *Cognate*(s) and $B \rightarrow \rho$ is numbered i then Reduce $i\}$
  - $\{$If $A \rightarrow \alpha \bullet a\beta \in s$ 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 → T •, {), $, +}
T → T • * P, {), $, +, *}

State 11:

E → E + T •, {), $, +}
T → T • * P, {), $, +, *}

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

Action table (same as SLR(1))

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

State	Lookahead					
	+	*	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);	⇒	DOWHILE(I) = 1 ;
Turing	gut S	⇒	**get** S
		⇒	**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.

Summary LL(k) and LR(k) Parsing

- A parser which can determine the correct derivation (reduction) at each stage as it scans the input stream in one direction is called *deterministic.*
- A grammar is LL(k) if a parser can parse strings scanning from left to right using *left* derivations with k-symbol lookahead deterministically.
- A grammar is LR(k) if a parser can parse strings scanning from left to right using *right* reductions with k-symbol lookahead deterministically.
- LR(k) is the most general deterministic left-to-right language.
  Language properties:

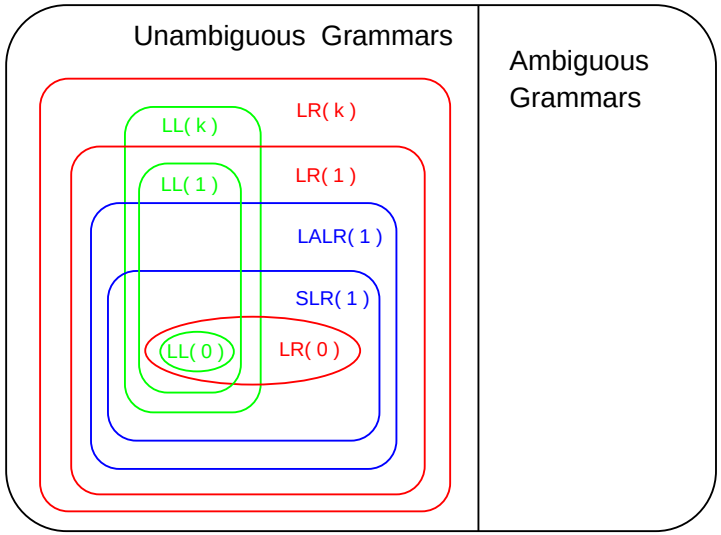
  $\mathcal{L}$(LL(k)) $\subset$ $\mathcal{L}$(LR(1))

  $\mathcal{L}$(LR(1)) $\subset$ $\mathcal{L}$(LR(k))

  $\mathcal{L}$(LL(k)) $\subset$ $\mathcal{L}$(LR(k))

  There are LR(0) grammars which are not LL(k) for any k.

Hierarchy of Grammar Classes[a]



[a] Adapted from, Andrew Appel, *Modern Compiler Implementation in Java, 2002*

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 have no parser generator, try to make the grammar LL(1) and then build an LL(1) parser or use recursive descent.
- 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[a]

[a] For example Edison Design Group `www.edg.com`

YACC, Bison, JCup, et. al.

- YACC is a widely available LALR(1) parser generator developed at Bell Labs. Bison is a freeware clone from GNU. Many other variants have been developed.
- YACC assumes input from Lex (or flex)lexical analyzer
- YACC allows the compiler writer to provide an arbitrary piece of C code for each rule in the grammar. This code gets executed just before the parser performs a reduction involving the rule. There is a convention for accessing information stored parallel to the parse stack. YACC does automatic rule splitting to handle code that is not at the end of a rule.
- YACC processes a grammar and produces tables that are used with a fixed parsing algorithm. The pieces of C code become the body of a giant switch statement.
- jflex, jcup – versions of flex and bison for Java.