

Reprise on Languages and Grammars

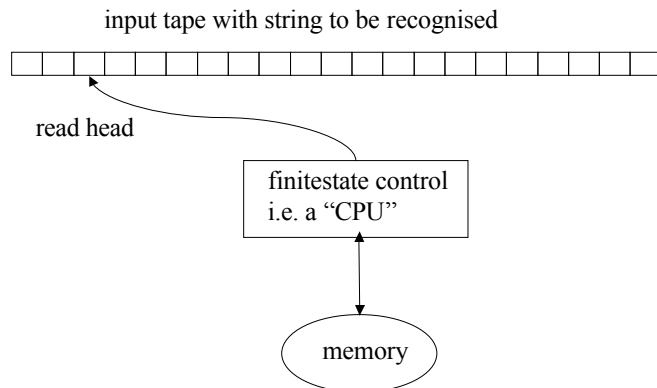
- A *language* is a set (usually infinite) of strings, also known as *sentences*
- Each string consists of a sequence of symbols taken from some *alphabet*
- You cannot define a language by listing the strings
- You can define a language by a *grammar* — a finite set of rules for generating the strings
- A grammar, G, is a quadruple (V_N, V_T, P, S) where
 - V_N is a set of symbols called *nonterminals*
 - V_T is a set of symbols called *Terminals*, the "words" of the language
 - P is a set of *Productions* — the rules for the formation of sentences
 - $S \in V_N$ is the *Start symbol* — the starting point.

Different kinds of Grammars

- Grammars have a set of *productions*
- Most general form is $\alpha \rightarrow \beta$
- Where α is a string including a non-terminal and β is any string (of terminals and non-terminals)
- Different kinds of grammars constrain the productions:
 - TYPE 1: $|\alpha| \leq |\beta|$
 - TYPE 2: α consists only of a non-terminal
 - TYPE 3: α consists only of a non-terminal and β consists only of a terminal or a terminal followed by a non-terminal

Reprise on Languages and Machines

- We can also define a language by specifying a machine that will recognise sentences of the language.
- The general structure of these machines is:



Different kinds of machines

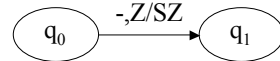
- The kind of auxiliary memory determines the kinds of languages that can be recognised:

Aux memory	Kind of machine	Kind of grammar
None	Finite state aut.	TYPE3
Stack	Push down aut.	TYPE2
Tape bounded by input length	Linear bounded aut.	TYPE1
Unbounded tape	Turing machine	TYPE0

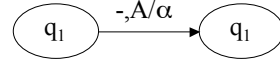
Key result for Push Down Automata

- The class of languages that can be recognised by push down automata is the same as the class of languages that can be generated by TYPE2 grammars
- This can be proved/demonstrated by showing how to map between the two kinds of specifications
- Consider a TYPE2 grammar with start symbol S :
 - introduce an initial state q_0 , a processing state q_1 and a final state q_2

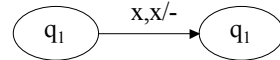
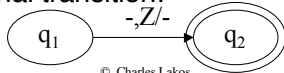
- for start symbol S , add a transition
- for productions $A \rightarrow \alpha$, add a transition:



- for terminals x , add a transition:

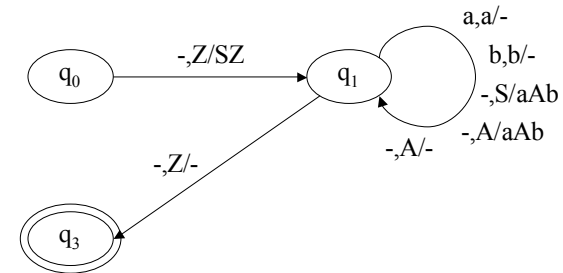


- add a final transition:



Example

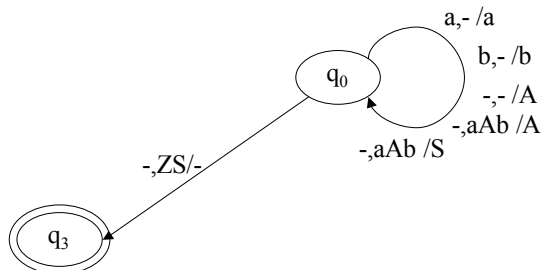
- Consider the earlier example of the language $a^n b^n$ where $n \geq 1$
- Consider the grammar:
 - $\{S, A\}$, $\{a, b\}$, P , S
- where P consists of the productions:
 - $S \rightarrow aAb$ $A \rightarrow \epsilon$ $A \rightarrow aAb$
- The push down automaton would be:



5 labels!
5 arcs!

Example

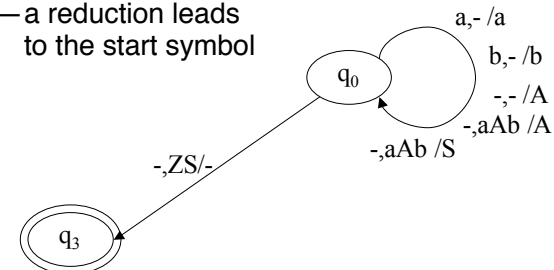
- Consider the earlier example of the language $a^n b^n$ where $n \geq 1$ and different kind of stack machine
- Consider the grammar:
 - $\{S, A\}$, $\{a, b\}$, P , S
- where P consists of the productions:
 - $S \rightarrow aAb$ $A \rightarrow \epsilon$ $A \rightarrow aAb$
- The push down automaton would be:



5 labels!
5 arcs!

Extended Pushdown Automata - EPDA

- The above stack machine is different in having the top of stack on the right (as opposed to the left)
- It is also different in removing multiple symbols off the stack at once — hence the name “Extended”
- It essentially parses in a bottom up fashion
 - the right hand sides of productions are accumulated on the stack
 - a reduction leads to the start symbol



5 labels!
5 arcs!

Example of EPDA

- We can demonstrate acceptance of the string aaabbb with a sequence of *configurations*, each of which indicates the current state, the unprocessed input and the stack contents (top of stack on the right!!):

$(q_0, aaabbb, Z)$	-	$(q_0, aaabbb, Za)$	
	-	$(q_0, abbb, Zaa)$	
	-	$(q_0, bbb, Zaaa)$	
	-	$(q_0, bbb, ZaaaA)$	# used $A \rightarrow \epsilon$
	-	$(q_0, bb, ZaaaAb)$	
	-	$(q_0, bb, ZaaA)$	# used $A \rightarrow aAb$
	-	$(q_0, b, ZaaAb)$	
	-	$(q_0, b, ZaaA)$	# used $A \rightarrow aAb$
	-	$(q_0, -, ZaAb)$	
	-	$(q_0, -, ZS)$	# used $S \rightarrow aAb$
	-	$(q_1, -, -)$	

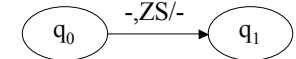
Accept!

Key result for Extended PDA

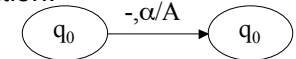
- The class of languages that can be recognised by extended push down automata is the same as the class of languages that can be generated by TYPE2 grammars

- This can be proved/demonstrated by showing how to map between the two kinds of specifications
- Consider a TYPE2 grammar with start symbol S:

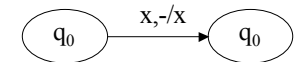
- introduce an initial state q_0 , a final state q_1
- for start symbol S, add a transition



- for productions $A \rightarrow \alpha$, add a transition:



- for terminals x, add a transition:



Kind of parse

- In the PDA model, the parse is top down
 - you start by putting the goal symbol on the TOS
 - if a non-terminal is on the TOS, you expand it by some production alternative
 - eventually the terminals on the stack match the input
- PDA model follows a left-most derivation
- In the EPDA model, the parse is bottom up
 - you push the terminals onto the stack (always possible)
 - you reduce the string on the TOS to a non-terminal
 - eventually, you have the goal symbol on the TOS
- EPDA model follows the reverse of a right-most derivation

Non-determinism

- In the PDA model, the non-deterministic choice is to choose a possible expansion for a non-terminal
 - with a non-terminal on top of stack
 - which alternative expansion to choose?
- In the EPDA model, the non-deterministic choice is whether to shift or reduce, and which reduction to use
 - do you push a terminal onto the stack (always possible)
 - if the top of stack string matches the right hand side of a production, do we reduce by that production?
 - what if there is more than one alternative?

Deterministic bottom-up parsing

- In order to achieve deterministic parsing decisions, we intersperse the terminals and non-terminals on the stack with symbols that indicate a more precise context information
- Then the parsing action tables are given in the form:
 - Action(X,a) = shift n / reduce n / accept / error
 - Goto(X,Y) = symbol to push onto stack after reduction
 - X = TOS symbol, a = input terminal
 - shift n = shift terminal a + symbol n onto the stack
 - reduce n = reduce by production n
 - accept / error = accept string / report a parsing error
 - Y = non-terminal on LHS of applied production

Example: Expressions

	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		

$S \rightarrow E$ 0
 $E \rightarrow E+T$ 1
 T 2
 $T \rightarrow T*F$ 3
 F 4
 $F \rightarrow (E)$ 5
 id 6

Example: Expressions

- Consider parsing: $x + x * x$
- (0, $x+x*x$)
 - l- (0x5, $+x*x$) # action was s5
 - l- (0F3, $+x*x$) # action was r6
 - l- (0T2, $+x*x$) # action was r4
 - l- (0E1, $+x*x$) # action was r2
 - l- (0E1+6, $x*x$) # action was s6
 - l- (0E1+6x5, $*x$) # action was s5
 - l- (0E1+6F3, $*x$) # action was r6
 - l- (0E1+6T9, $*x$) # action was r4
 - l- (0E1+6T9*7, x) # action was s7
 - l- (0E1+6T9*7x5, $)$ # action was s5
 - l- (0E1+6T9*7F10, $)$ # action was r6
 - l- (0E1+6T9, $)$ # action was r3
 - l- (0E1, $)$ # action was 1
 - l- accept

Example: Expressions

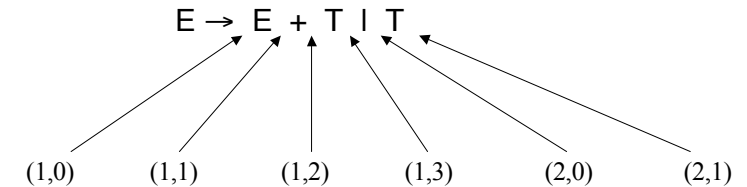
- Note that applying a reduction is a complex operation:
 - For: (0E1+6T9*7F10, $)$ action is r3
 - Production 3 is $T \rightarrow T * F$
 - This production has 3 symbols on the RHS
 - We need to remove 6 symbols from the stack
 - the grammar symbols and the corresponding stack symbols
 - This leaves 0E1+6 on the stack with symbol 6 on top
 - The LHS of the production is T
 - The entry goto(6,T) indicates 9
 - So, the stack becomes 0E1+6T9

Observations

- The parse was deterministic without changing the grammar
 - bottom-up parsing can cope with more grammars (and languages) without change than top-down parsing
- The table is quite large by comparison
 - attempts to reduce the table size differentiate the various flavours of bottom up parsing algorithms
- The question is what do the stack symbols represent and how do we generate them?

Bottom-up parsing (1)

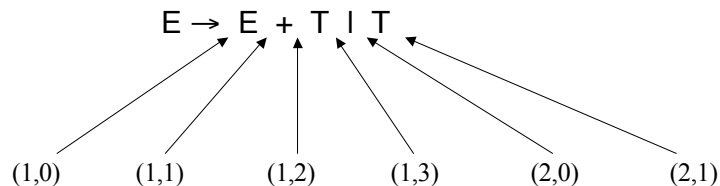
- The stack symbols keep track of the possible positions in productions



- A symbol on the stack indicating position (1,1) tells us that '+' is a valid symbol and can be pushed onto the stack which takes us to position (1,2)

Bottom-up parsing (2)

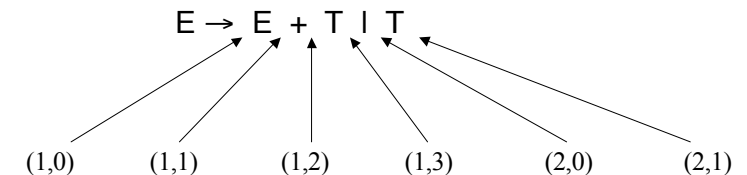
- The stack symbols keep track of the possible sets of positions in productions



- Position (1,0) cannot be considered in isolation
 - we need to consider position (2,0) as well
 - the start of an E could equally well be the start of a T
- This is known as a **closure** operation

Bottom-up parsing (3)

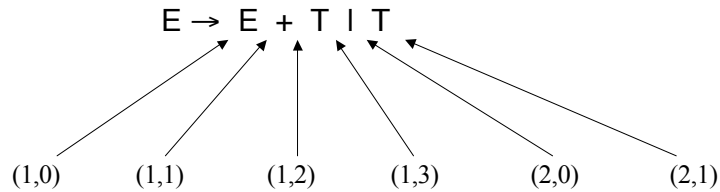
- The stack symbols keep track of the possible sets of positions in productions **together with the possible follower symbols**



- If the goal symbol is E, which is followed by \$ (EOF), then we start with position (0,0) with \$, i.e. (0,0,\$)
- Closure will add items (1,0,\$), (2,0,\$)
- Closure will add items (1,0,+), (2,0,+) — expansions of E with + following — giving us (0,0,\$), (1,0,\$+), (2,0,\$+)

Bottom-up parsing (4)

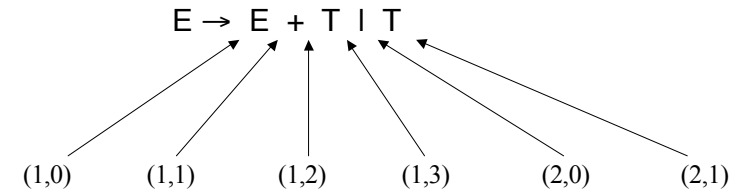
- The stack symbols keep track of the possible sets of positions in productions and the follower symbols



- Positions at end of productions will have associated **reduce** actions for each of the follower symbols
—e.g. $(1,3,\$+)$ will have “r1” actions for $\$$ and $+$

Bottom-up parsing (5)

- The stack symbols keep track of the possible sets of positions in productions and the follower symbols



- Positions in the middle of productions will have **shift** actions (terminals) and **goto** entries (non-terminals)
—e.g. $(1,1,\$+)$ will have a shift entry for $+$ to a symbol which has $(1,2,\$+)$ and its closure
—e.g. $(1,0,\$+)$ will have a goto entry for E to a symbol which has $(1,1,\$+)$ and its closure

Example: Expressions

	Action						Goto		
	id	+	*	()	\$	E	T	F
0 = $(0,0,\$)$, $(1,0,\$+)$, $(2,0,\$+)$, $(3,0,\$+^*)$, $(4,0,\$+^*)$, $(5,0,\$+^*)$, $(6,0,\$+^*)$	s5				s4		1	2	3
1 = $(0,1,\$)$, $(1,1,\$+)$		s6				acc			
2 = $(2,1,\$+)$, $(3,1,\$+^*)$		r2	s7			r2			
3 = $(4,1,\$+^*)$		r4	r4			r4			
4 = $(5,1,\$+^*)$, $(1,0,+)$, $(2,0,+)$, $(3,0,+^*)$, $(4,0,+^*)$, $(5,0,+^*)$, $(6,0,+^*)$	s10				s9		8	2	3
5 = $(6,1,\$+^*)$		r6	r6			r6			
6 = $(1,2,\$+)$, $(3,0,\$+^*)$, $(4,0,\$+^*)$, $(5,0,\$+^*)$, $(6,0,\$+^*)$	s5				s4			11	3
7 = $(3,2,\$+^*)$, $(5,0,\$+^*)$, $(6,0,\$+^*)$	s5				s4				12
8 = $(5,2,\$+^*)$, $(1,1,+)$		s14			s13				
9 = $(5,1,+^*)$, $(1,0,+)$, $(2,0,+)$, $(3,0,+^*)$, $(4,0,+^*)$, $(5,0,+^*)$, $(6,0,+^*)$	s10				s9		15	16	17

Example: Expressions

	Action						Goto		
	id	+	*	()	\$	E	T	F
10 = $(6,1,+^*)$		r6	r6			r6			
11 = $(1,3,\$+)$, $(3,1,\$+^*)$		r1	s7			r1			
12 = $(3,3,\$+^*)$		r3	r3			r3			
13 = $(5,3,\$+^*)$		r5	r5			r5			
14 = $(1,2,+)$, $(3,0,+^*)$, $(4,0,+^*)$, $(5,0,+^*)$, $(6,0,+^*)$	s10				s9			18	17
15 = $(5,2,+^*)$, $(1,1,+)$		s14				s19			
16 = $(2,1,+)$, $(3,1,+^*)$		r2	s20			r2			
17 = $(4,1,+^*)$		r4	r4			r4			
18 = $(1,3,+)$, $(3,1,+^*)$		r1	s20			r1			
19 = $(5,3,+^*)$		r5	r5			r5			
20 = $(3,2,+^*)$, $(5,0,+^*)$, $(6,0,+^*)$	s10				s9				21
21 = $(3,3,+^*)$		r3	r3			r3			

LR parsing table construction (1)

```

type s_items = set of items;
function closure(I : s_items) : s_items;
  var temp, res : s_items;
  begin
  res := I;
  repeat
    temp := res;
    for all (A → α.Bβ, a) ∈ temp do      # '.' indicates position in production
    for all productions B → γ do        # consider expansions of B
    for all terminals b ∈ FIRST(βa) do   # what can follow B
      res := res + (B → .γ, b);        # possibly new items added to set
  until temp = res;
  return res;
end;

```

LR parsing table construction (2)

```

var X1(=S), X2, ... Xn : non-terminals;
    Xn+1, Xn+2, ... Xm : terminals;
    temp, I1, I2, ... : s_items;
begin
i := 0; p := 1; I1 := closure({(S' → .S, $)}); # start here
while i < p do # while sets to process
  begin
  i := i+1;
  for all (A → β., Xj) ∈ Ii where A → β is production k do
    if (A → β., Xj) is (S' → S., $) then
      action[i,j] := accept else
      action[i,j] := reduce k
  ...

```

LR parsing table construction (3)

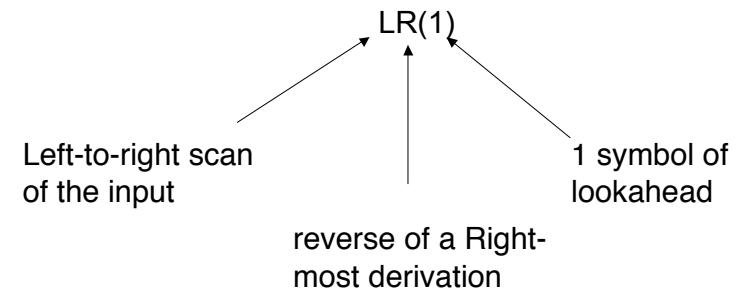
```

while i < p do # while sets to process
  ...
  for all terminal/non-terminal Xj do
  begin
  temp := closure({all (A → αXj.β, a)}) where (A → αXjβ, a) ∈ Ii
  if not isempty(temp) then
  begin
  Ip+1 := temp; # set up sentinel
  k := minimum r such that Ir = temp; # find set of items - or sentinel
  if k = p+1 then p := p+1; # increment p for new set of items
  if Xj is terminal then # setting table entry
    action[i,j] = shift k else
    goto[i,j] := k;
  end
  end
end
end

```

LR parsers

- The above techniques are generically referred to as LR(1) parser



LR parsers

- Specifically, there are differences in LR(1) parsers:
- LR(1) parsers differentiate stack symbols by both the positions and the follower symbols
- LALR(1) parsers merge stack symbols if they represent the same set of positions but different follower symbols (the sets of follower symbols are merged)
- SLR(1) only differentiates sets of positions and determines follower symbols using the algorithms presented earlier in the course

LALR(1) parsers

- LALR(1) parsers merge stack symbols if they represent the same set of positions but different follower symbols (the sets of follower symbols are merged)
- In the expression parsing table, we merge the following symbols:
 - 2 and 16, 3 and 17, 4 and 9, 5 and 10, 6 and 14, 7 and 20, 8 and 15, 11 and 18, 12 and 21, 13 and 19
- This gives us the table overleaf which is identical to the one first considered

Example: Expressions

	Action						Goto		
	id	+	*	()	\$	E	T	F
0 = (0,0,\$), (1,0,\$+), (2,0,\$+), (3,0,\$+*), (4,0,\$+*), (5,0,\$+*), (6,0,\$+*)	s5			s4			1	2	3
1 = (0,1,\$), (1,1,\$+)		s6				acc			
2 = (2,1,\$+), (3,1,\$+*)		r2	s7		r2	r2			
3 = (4,1,\$+*)		r4	r4		r4	r4			
4 = (5,1,\$+*), (1,0,+), (2,0,+), (3,0,+*), (4,0,+*), (5,0,+*), (6,0,+*)	s5			s4			8	2	3
5 = (6,1,\$+*)		r6	r6		r6	r6			
6 = (1,2,\$+), (3,0,\$+*), (4,0,\$+*), (5,0,\$+*), (6,0,\$+*)	s5			s4				9	3
7 = (3,2,\$+*), (5,0,\$+*), (6,0,\$+*)	s5			s4					10
8 = (5,2,\$+*), (1,1,+)		s6			s11				
9 = (1,3,\$+), (3,1,\$+*)		r1	s7		r1	r1			
10 = (3,3,\$+*)		r3	r3		r3	r3			
11 = (5,3,\$+*)		r5	r5		r5	r5			

LR parsers

- LR(1) parsers are able to handle the largest class of grammars (without modification) but produce very large tables
- LALR(1) parsers are able to handle most common language grammars with a significant reduction in table size compared to LR(1)
- SLR(1) parsers have the smallest tables but may have problems with common language grammars
- Hence, LALR(1) is a common choice in parser-generators

Tricky grammars

- The following grammar is LALR(1) but not SLR(1)

$S \rightarrow L = R \mid R$ # assignment or expression
 $L \rightarrow * R \mid id$ # indirection or identifier
 $R \rightarrow L$ # R-value can be an L-value

- The following grammar is LR(1) but not LALR(1)

$A \rightarrow aAd \mid bBd \mid aBe \mid bAe$
 $A \rightarrow c$
 $B \rightarrow c$

Using an LR parser generator

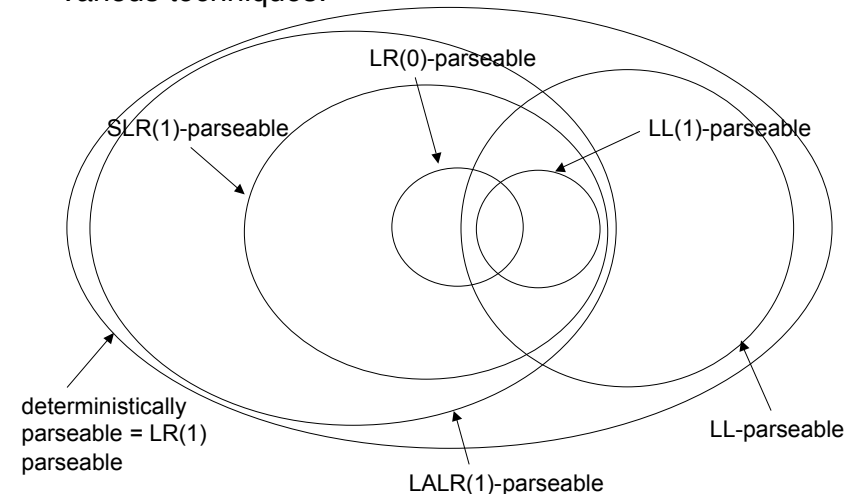
- You define your grammar in the appropriate notation and feed it to the parser generator
- It may report “shift-reduce conflicts”
 - the parser has certain situations when it cannot decide between a shift and a reduce action
 - default response is to prefer a shift over a reduce
- It may report “reduce-reduce conflicts”
 - the parser has certain situations where it cannot decide which production to use for a reduction
 - commonly occurs when you have empty productions
 - need to resolve the problem, possibly by factoring the grammar
- Typically, you encounter a handful of such situations

Properties of LR parsing

- The parse time is linear relative to the length of the input
- Grammars generally don't require modification, or at least only minor modification!!
- LR parsing has good error detection — no shift is performed if a symbol is incompatible with the grammar
 - erroneous input is reported as soon as possible
- Error recovery is not so simple
 - cf. LL parsing where the stack contains the unmatched part of the sentential form, and hence the context

Parseable languages

- We can compare the *languages* that can be parsed by various techniques:



Parseable languages

- The previous diagram is in terms of parseability of *languages*
- Just because you have a grammar which is not LR(1) doesn't mean that you can't find a grammar for the language which is LR(1)
- For all k , there are languages which are LL($k+1$)-parseable but not LL(k) parseable
- For all k , if a language is LR(k)-parseable, then it is also LR(1)-parseable