Bottom-up parsing

- LR(*k*) is a less restrictive subset of CFGs than LL(*k*)
  - left to right, rightmost reduction, *k* symbols of lookahead
  - it uses a bottom-up parser instead of top-down
- Donald Knuth proved that every LR(*k*) grammar can be rewritten as an LR(1) grammar
  - this means we don't need anything more complex than LR(1)

Bottom-up parsing

- Consider the following example grammar:
  - <statement> ::= <expression> ;
  - <expression> ::= <term> | <expression> + <term>
  - <term> ::= <number> | (<expression>)
- note that this is left recursive!
- to save space, I will abbreviate individual sub-rules like this:
  - 1. S ::= E ;
  - 2. E ::= T
  - 3. E ::= E + T
  - 4. T ::= n
  - 5. T ::= (E)

Bottom-up parsing

1. S ::= E ;
2. E ::= T
3. E ::= E + T
4. T ::= n
5. T ::= (E)

Input: n + (n) ;

- Only rule 4 starts with n
  - shift: we have n, we're looking at +

Bottom-up parsing

1. S ::= E ;
2. E ::= T
3. E ::= E + T
4. T ::= n
5. T ::= (E)

Input: n + (n) ;

- No rule starts with + or n+
  - the only way to continue is to reduce n to T (rule 4)

Bottom-up parsing

1. S ::= E ;
2. E ::= T
3. E ::= E + T
4. T ::= n
5. T ::= (E)

Input: T + (n) ;

- No rule starts with + or T+
  - the only way to continue is to reduce T to E (rule 4)

Bottom-up parsing

1. S ::= E ;
2. E ::= T
3. E ::= E + T
4. T ::= n
5. T ::= (E)

Input: E + (n) ;

- Only rule 3 starts with E+
  - shift: we have E +, we're looking at (

Bottom-up parsing

1. S ::= E ;
2. E ::= T
3. E ::= E + T
4. T ::= n
5. T ::= (E)

Input: E + (n) ;

- Only rule 5 starts with (
  - shift: we have E + (, we're looking at n

Bottom-up parsing

1. S ::= E ;
2. E ::= T
3. E ::= E + T
4. T ::= n
5. T ::= (E)

Input: E + (n) ;

- Only rule 4 starts with n
  - shift: we have E + (n, we're looking at)

Bottom-up parsing

1. S ::= E ;
2. E ::= T
3. E ::= E + T
4. T ::= n
5. T ::= (E)

Input: E + (n) ;

- No rule starts with) or (n or E+(n
  - the only way to continue is to reduce n to T (rule 4)

Bottom-up parsing

1. S ::= E ;
2. E ::= T
3. E ::= E + T
4. T ::= n
5. T ::= (E)

Input: E + (T) ;

- No rule starts with) or (T or E+(T
  - the only way to continue is to reduce T to E (rule 2)

Bottom-up parsing

1. S ::= E ;
2. E ::= T
3. E ::= E + T
4. T ::= n
5. T ::= (E)

Input: E + (E) ;

- Rule 5 starts with (E)
  - shift: we have E+(E), we're looking at ;

Bottom-up parsing

1. S ::= E ;
2. E ::= T
3. E ::= E + T
4. T ::= n
5. T ::= (E)

Input: E + (E) ;

- Rule 5 matches (E)
  - the only way to continue is to reduce (E) to T (rule 5)

Bottom-up parsing

1. S ::= E ;
2. E ::= T
3. E ::= E + T
4. T ::= n
5. T ::= (E)

Input: E + T ;

- Rule 3 matches E + T
  - continue by reducing E+T to E (rule 3)

Bottom-up parsing

1. S ::= E ;
2. E ::= T
3. E ::= E + T
4. T ::= n
5. T ::= (E)

Input: E ;

- Rule 1 starts with E
  - shift, we're at the end of the input
- Rule 1 matches E;
  - reduce to S, and we're finished!

Bottom-up parsing

- LR parsers are table-driven
  - the grammar is transformed into a table:

– we will see a short example later showing how the table is used to parse input

	n	+	(	)	;	E	T
0	s5		s6			1	4
1		s2			a		
2	s5		s6				3
3		r2		r2	r2		
4		r3		r3	r3		
5		r4		r4	r4		
6	s5		s6			7	4
7		s2		s8			
8		r5		r5	r5		

Building LR tables

- To build the table, start with the rule for the target state S
  - this will become state 0 in the table
  - mark the initial parser position in the rule with a dot:
    - $S \to \bullet E \;;$
  - this means that the parser needs to recognise an E next, so repeat the process for each of the rules for E:
    - $E \to \bullet E + T$ $\qquad$ $E \to \bullet T$

Building LR tables

- Because E can start with a T, do the same for the rules for T:
  - $T \rightarrow \bullet n$
  - $T \rightarrow \bullet (E)$
  - this completes state 0:
    - $S \rightarrow \bullet E \;;$
    - $E \rightarrow \bullet E + T$
    - $E \rightarrow \bullet T$
    - $T \rightarrow \bullet n$
    - $T \rightarrow \bullet (E)$

Building LR tables

- From state 0:
    - $S \rightarrow \bullet E\ ;$
    - $E \rightarrow \bullet E + T$
    - $E \rightarrow \bullet T$
    - $T \rightarrow \bullet n$
    - $T \rightarrow \bullet (E)$
  - we can continue if we see E, T, n or (
  - we need to construct states for what happens after we see each of these

Building LR tables

- After E (state 1) we are in one of two places:
  - $S \rightarrow E \bullet ;$
  - $E \rightarrow E \bullet + T$
  - the next token must be either ‘;’ or ‘+’
  - if the next token is ‘;’ the parse succeeds (accept)
  - now we construct state 2 showing where we are after matching ‘+’

Building LR tables

- State 2 starts after ‘+’ in rule 1:
    - $E \rightarrow E + \bullet T$
  - since we are at the start of a T, we need to include the rules for T:
    - $T \rightarrow \bullet n$
    - $T \rightarrow \bullet(E)$
  - what comes next must be T, ‘n’ or ‘(’
  - the rules for T are also part of state 0, so we can deal with what happens after ‘n’ or ‘(’ the same way as state 0 does

Building LR tables

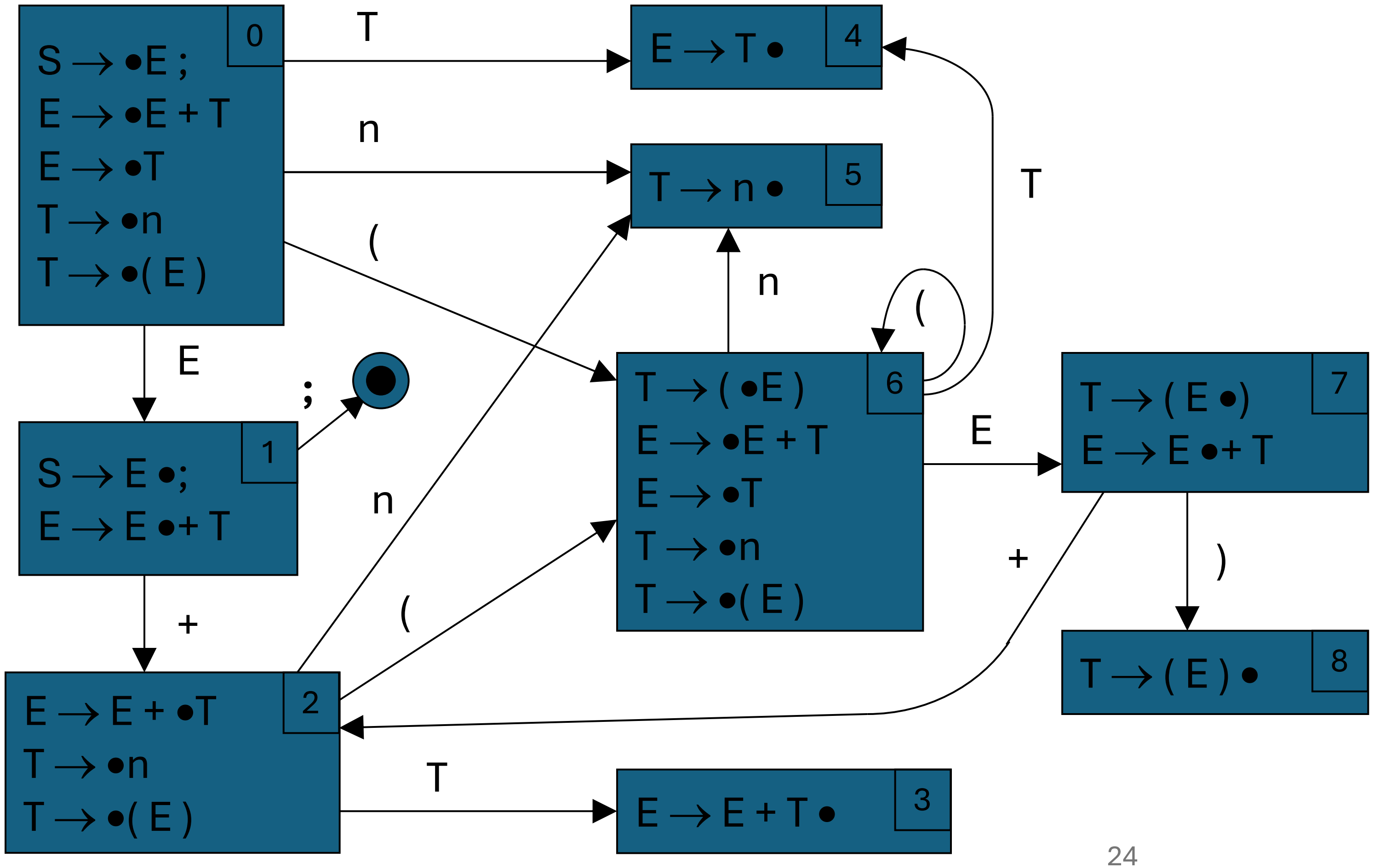
- State 3: we are at the end of rule 2
  - $E \rightarrow E + T \bullet$
  - we now reduce E + T to E
- State 4: continue after the T in rule 3
  - $E \rightarrow T \bullet$
  - we now reduce T to E
- State 5: continue after the ‘n’ in rule 4
  - $T \rightarrow n \bullet$
  - we now reduce ‘n’ to T

Building LR tables

- State 6: after ‘(’ in rule 5
    - $T \rightarrow (\bullet E)$
  - include rules for E and T, as we did for state 0:
    - $E \rightarrow \bullet E + T$ $T \rightarrow \bullet n$
    - $E \rightarrow \bullet T$ $T \rightarrow \bullet (E)$
- State 7: after E in rule 5
    - $T \rightarrow (E \bullet)$
    - $E \rightarrow E \bullet + T$
- State 8: at the end of rule 5 (reduce to T)
    - $T \rightarrow (E) \bullet$

Building LR tables

- Here is the complete set of states:



Building LR tables

- For each state:
  - when the dot is in front of a terminal symbol, insert ‘shift *s*’ in the column for that terminal, where *s* is the state containing the same rule with the dot after the same terminal
  - when the dot is in front of a non-terminal symbol, put the state that will be reached when the non-terminal is matched in the column for that non-terminal (for example, in state 0, go to state 1 after matching E, go to state 4 after matching T)

Building LR tables

- For each state:
  - when the dot is at the end of a rule, for every terminal symbol that can follow the nonterminal symbol on the left of the rule, insert ‘reduce *n*’ in the column for that terminal, where *n* is the rule number in the original grammar

Bottom-up parsing

1. S ::= E ;
2. E ::= T
3. E ::= E + T
4. T ::= n
5. T ::= (E)

sN : shift: put token and N on the stack, go to state N

rN : reduce: remove right-hand side elements for rule N from the stack, go to state specified by entry on top of the stack and left-hand side of rule N, and put left-hand side of rule and new state on the stack

a : accept (parse succeeded)

blank : error

	n	+	(	)	;	E	T
0	**s5**		s6			1	4
1		s2			a		
2	s5		s6				3
3		r2		r2	r2		
4		r3		r3	r3		
5		r4		r4	r4		
6	s5		s6			7	4
7		s2		s8			
8		r5		r5	r5		

(See slide 46 for details of states and the transitions between them)

Parsing 'n + (n) ;'

1. $S \to E\ ;$
2. $E \to E + T$
3. $E \to T$
4. $T \to n$
5. $T \to (E)$

Stack: 0

Input: n + (n) ;

Action: 0/n = s5
(Shift n to stack, go to state 5,
push new state 5 onto stack)

	n	+	(	)	;	E	T
0	**s5**		s6			1	4
1		s2			a		
2	s5		s6				3
3		r2		r2	r2		
4		r3		r3	r3		
5		r4		r4	r4		
6	s5		s6			7	4
7		s2		s8			
8		r5		r5	r5		

Parsing ‘n + (n) ;’

1. S → E ;
2. E → E + T
3. E → T
4. T → n
5. T → (E)

Stack: 0 n 5

Input: + (n) ;

Action: 5/+ = r4
(Remove 5 and n from stack,
reduce n by rule 4 to T, go to state on top of stack (0),
push new symbol T and state 0/T = 4)

	n	+	(	)	;	E	T
0	s5		s6			1	**4**
1		s2			a		
2	s5		s6				3
3		r2		r2	r2		
4		r3		r3	r3		
5		**r4**		r4	r4		
6	s5		s6			7	4
7		s2		s8			
8		r5		r5	r5		

Parsing ‘n + (n) ;’

1. S → E ;
2. E → E + T
3. E → T
4. T → n
5. T → (E)

Stack: 0 T 4

Input: + (n) ;

Action: 4/+ = r3
(Remove 4 and T from stack,
reduce T by rule 3 to E, go to state on top of stack (0),
push new symbol E and state 0/E = 1)

	n	+	(	)	;	E	T
0	s5		s6			1	4
1		s2			a		
2	s5		s6				3
3		r2		r2	r2		
4		r3		r3	r3		
5		r4		r4	r4		
6	s5		s6			7	4
7		s2		s8			
8		r5		r5	r5		

Parsing ‘n + (n) ;’

1. $S \rightarrow E$;
2. $E \rightarrow E + T$
3. $E \rightarrow T$
4. $T \rightarrow n$
5. $T \rightarrow (E)$

Stack: 0 E 1

Input: + (n) ;

Action: 1/+ = s2
(Shift + to stack, go to state 2,
push new state 2 onto stack)

	n	+	(	)	;	E	T
0	s5		s6			1	4
1		**s2**			a		
2	s5		s6				3
3		r2		r2	r2		
4		r3		r3	r3		
5		r4		r4	r4		
6	s5		s6			7	4
7		s2		s8			
8		r5		r5	r5		

Parsing ‘n + (n) ;’

1. $S \rightarrow E$;
2. $E \rightarrow E + T$
3. $E \rightarrow T$
4. $T \rightarrow n$
5. $T \rightarrow (E)$

Stack: 0 E 1 + 2

Input: (n) ;

Action: 2/(= s6
(Shift (to stack, go to state 6,
push new state 6 onto stack)

	n	+	(	)	;	E	T
0	s5		s6			1	4
1		s2			a		
2	s5		**s6**				3
3		r2		r2	r2		
4		r3		r3	r3		
5		r4		r4	r4		
6	s5		s6			7	4
7		s2		s8			
8		r5		r5	r5		

Parsing ‘n + (n) ;’

1. S → E ;
2. E → E + T
3. E → T
4. T → n
5. T → (E)

Stack: 0 E 1 + 2 (6

Input: n) ;

Action: 6/n = s5
(Shift n to stack, go to state 5,
 push new state 5 onto stack)

	n	+	(	)	;	E	T
0	s5		s6			1	4
1		s2			a		
2	s5		s6				3
3		r2		r2	r2		
4		r3		r3	r3		
5		r4		r4	r4		
6	**s5**		s6			7	4
7		s2		s8			
8		r5		r5	r5		

Parsing 'n + (n) ;'

1. S → E ;
2. E → E + T
3. E → T
4. T → n
5. T → (E)

Stack: 0 E 1 + 2 (6 n 5

Input:) ;

Action: 5/) = r4
(Remove 5 and n from stack,
reduce n by rule 4 to T, go to state on top of stack (6),
push new symbol T and state 6/T = 4)

	n	+	(	)	;	E	T
0	s5		s6			1	4
1		s2			a		
2	s5		s6				3
3		r2		r2	r2		
4		r3		r3	r3		
5		r4		**r4**	r4		
6	s5		s6			7	**4**
7		s2		s8			
8		r5		r5	r5		

Parsing ‘n + (n) ;’

1. S → E ;
2. E → E + T
3. E → T
4. T → n
5. T → (E)

Stack: 0 E 1 + 2 (6 T 4

Input:) ;

Action: 4/) = r3
(Remove T 4 from stack,
reduce T by rule 3 to E, go to state on top of stack (6),
push new symbol E and state 6/E = 7)

	n	+	(	)	;	E	T
0	s5		s6			1	4
1		s2			a		
2	s5		s6				3
3		r2		r2	r2		
4		r3		**r3**	r3		
5		r4		r4	r4		
6	s5		s6			**7**	4
7		s2		s8			
8		r5		r5	r5		

Parsing ‘n + (n) ;’

1. S → E ;
2. E → E + T
3. E → T
4. T → n
5. T → (E)

Stack: 0 E 1 + 2 (6 E 7

Input:) ;

Action: 7/) = s8
(Shift) to stack, go to state 8,
push new state 8 onto stack)

	n	+	(	)	;	E	T
0	s5		s6			1	4
1		s2			a		
2	s5		s6				3
3		r2		r2	r2		
4		r3		r3	r3		
5		r4		r4	r4		
6	s5		s6			7	4
7		s2		**s8**			
8		r5		r5	r5		

Parsing ‘n + (n) ;’

1. S → E ;
2. E → E + T
3. E → T
4. T → n
5. T → (E)

Stack: 0 E 1 + 2 (6 E 7) 8

Input: ;

Action: 8/; = r5
(Remove (6 E 7) 8 from stack,
reduce (E) by rule 5 to T, go to state on top of stack (2),
push new symbol T and state 2/T = 3)

	n	+	(	)	;	E	T
0	s5		s6			1	4
1		s2			a		
2	s5		s6				**3**
3		r2		r2	r2		
4		r3		r3	r3		
5		r4		r4	r4		
6	s5		s6			7	4
7		s2		s8			
8		r5		r5	**r5**		

Parsing ‘n + (n) ;’

1. S → E ;
2. E → E + T
3. E → T
4. T → n
5. T → (E)

Stack: 0 E 1 + 2 T 3

Input: ;

Action: 3/; = r2
(Remove E 1 + 2 T 3 from stack, reduce E + T by rule 2 to E, go to state on top of stack (0), push new symbol E and state 0/E = 1)

	n	+	(	)	;	E	T
0	s5		s6			**1**	4
1		s2			a		
2	s5		s6				3
3		r2		r2	**r2**		
4		r3		r3	r3		
5		r4		r4	r4		
6	s5		s6			7	4
7		s2		s8			
8		r5		r5	r5		

Parsing 'n + (n) ;'

1. S $\rightarrow$ E ;
2. E $\rightarrow$ E + T
3. E $\rightarrow$ T
4. T $\rightarrow$ n
5. T $\rightarrow$ (E)

Stack: 0 E 1

Input: ;

Action: 1/; = a
(Accept; the input was parsed successfully)

	n	+	(	)	;	E	T
0	s5		s6			1	4
1		s2			**a**		
2	s5		s6				3
3		r2		r2	r2		
4		r3		r3	r3		
5		r4		r4	r4		
6	s5		s6			7	4
7		s2		s8			
8		r5		r5	r5		

LR conflicts

- Attempting to generate an LR parser from an ambiguous grammar will result in *conflicts*
  - *shift-reduce* conflicts: a particular table entry corresponds to both a shift action and a reduce action
  - *reduce-reduce* conflict: a particular table entry corresponds to more than one reduction

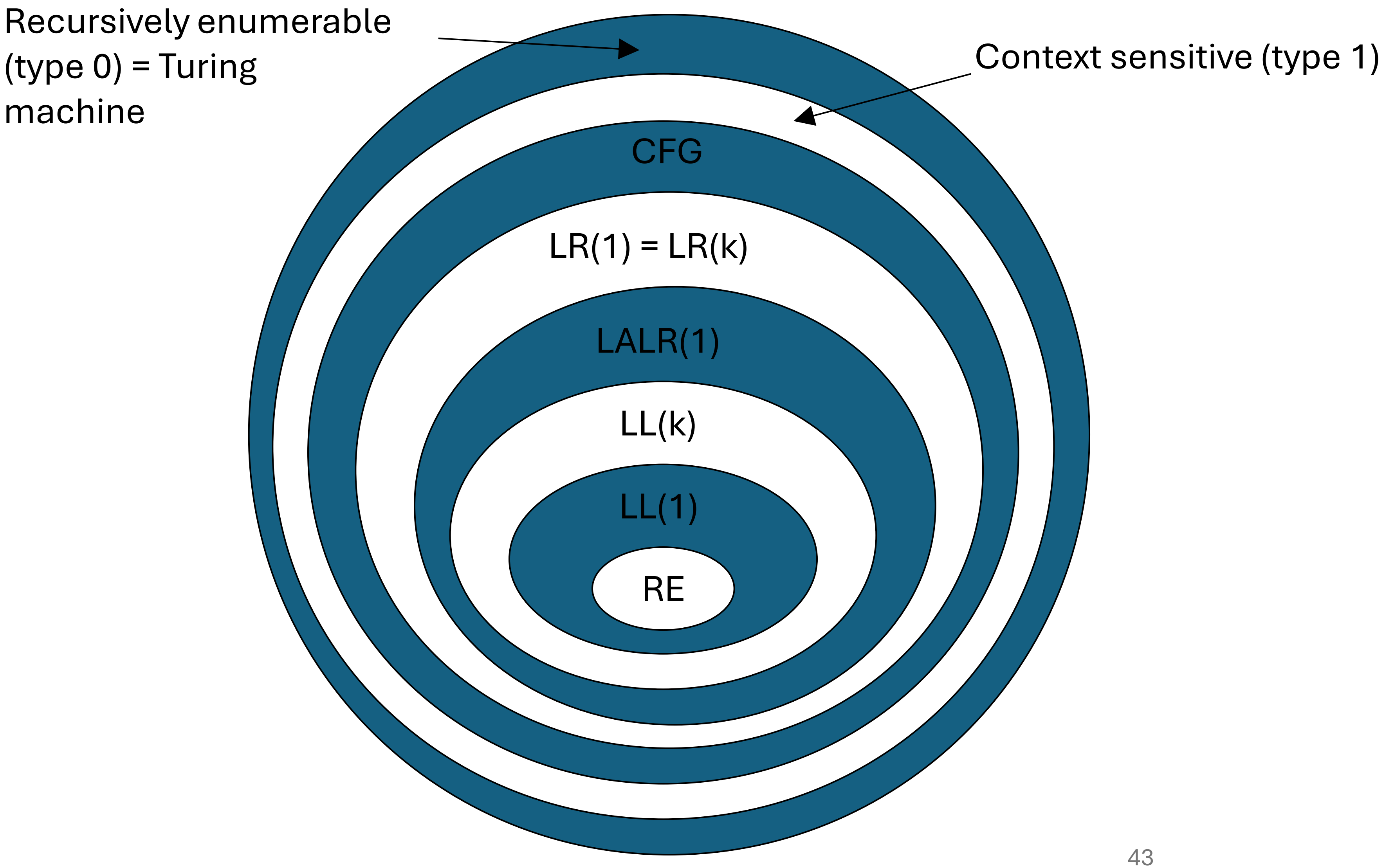
LR conflicts

- For ambiguous grammars:
  - rewrite the grammar to avoid ambiguity
  - tell the parser adopt a preferred interpretation (shift vs. reduce, or a preferred reduction) to resolve the ambiguity
  - for example, shift instead of reducing if there is a shift-reduce conflict, and reduce using the first of the conflicting rules that was declared if there is a reduce-reduce conflict

Building LR tables

- Building LR tables is hard work!
  - but it’s a mechanical process
  - tools exist to do it automatically
  - we will use a tool called JavaCUP to do the work for us…
- JavaCUP generates an LALR(1) parser
  - a subset of LR(1) which uses smaller tables

Grammar hierarchy



Coming next

- JFlex and JavaCUP
  - tools to build lexical and syntax analysers
- Code generation
- Optimisation
- Loose ends