Norwegian University of Science and Technology

Compiler Construction

Lecture 8: LR-parsing

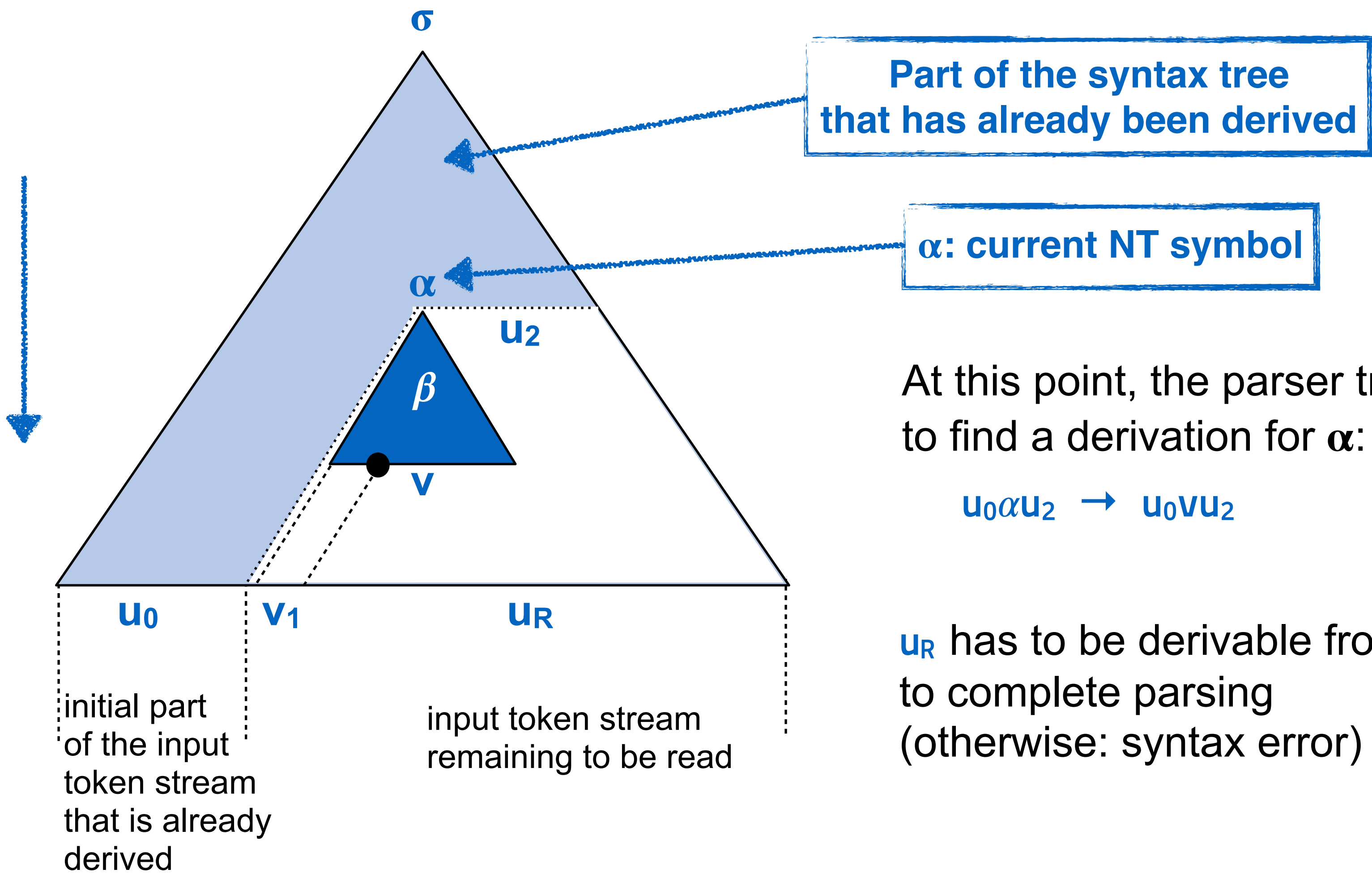
Michael Engel

Overview

- Bottom-up parsing revisited
- Deciding when to reduce
- LR parsers
  - General idea
  - LR(1) parsers
  - LALR

Top-down parsing

LL(1) parsers generate a parse tree from top to bottom:



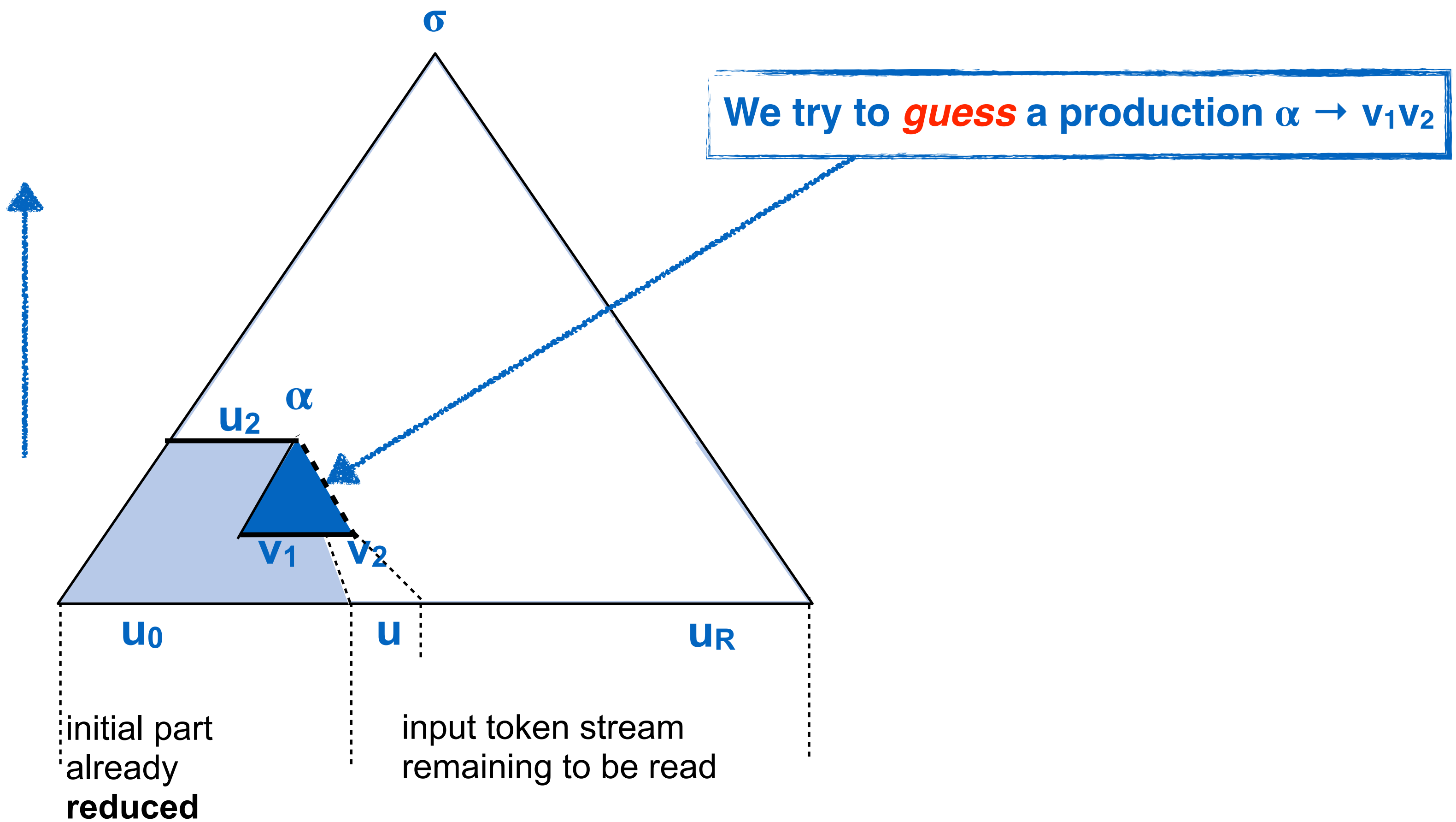
At this point, the parser tries to find a derivation for **α**:

$u_0 \alpha u_2 \rightarrow u_0 v u_2$

u_R has to be derivable from u_2 to complete parsing (otherwise: syntax error)

Bottom-up parsing

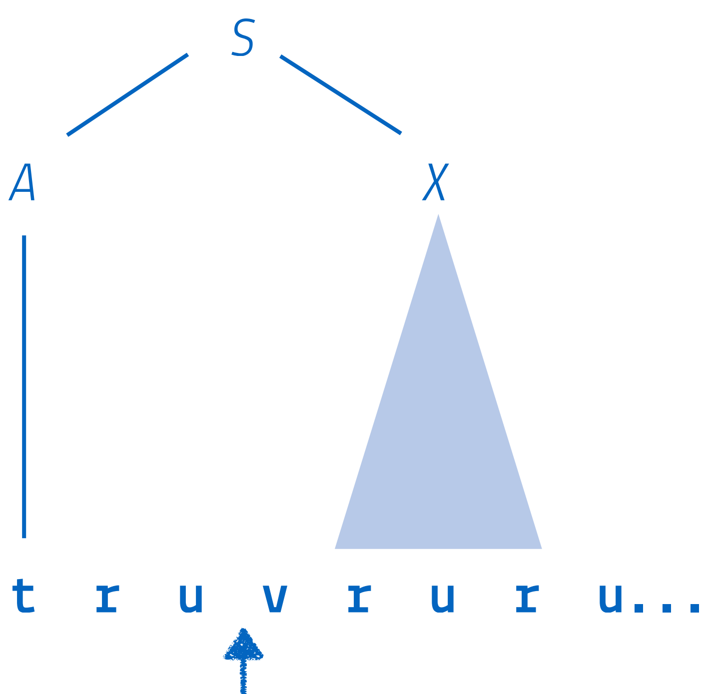
Can we also construct the parse tree from bottom to top?



Parsing compared in detail

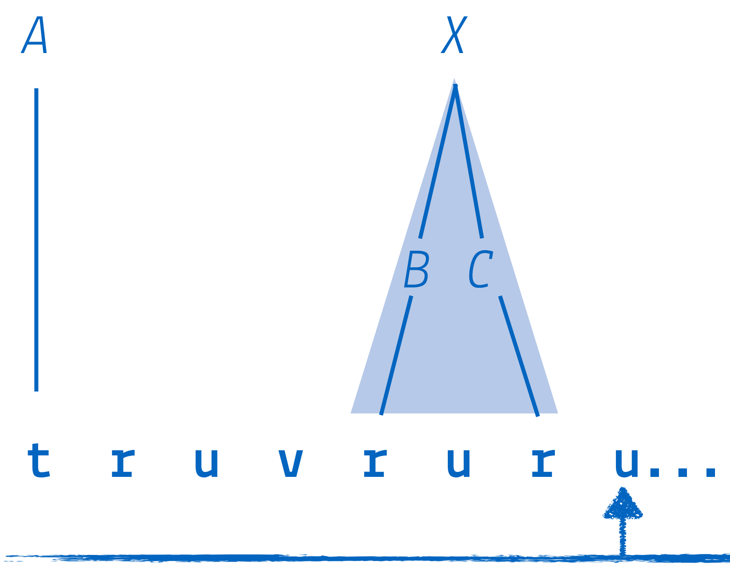
Syntax analysis

Top-down and bottom-up parsing and syntax tree construction



Top-down parsing / LL(1):
decides to build X after seeing the first token **of** its subtree

Botton-up parsing creates partial subtrees (a "forest") for parts of the input streams it can reduce



Botton-up parsing / LR(1):
decides to build X after seeing the first token **following** its subtree

General idea of bottom-up parsing

- Bottom-up parsing starts from the input token stream (whereas top-down starts from the grammar start symbol)
- It ***reduces*** a string to the start symbol by ***inverting productions***
  - trying to find a production matching the ***right hand side***

$E \rightarrow T + E \mid T$

$T \rightarrow$ int $\times\ T \mid$ int $\mid \varepsilon$

$E \leftarrow T + E \mid T$

$T \leftarrow$ int $\times\ T \mid$ int $\mid \varepsilon$

- Consider the input token stream **int * int + int**:

- Reading the productions in reverse (from ***bottom*** to ***top***) gives a ***rightmost derivation***

int × int + int	$T \rightarrow$ int
int × T + int	$T \rightarrow$ int × T
T + int	$T \rightarrow$ int
$T + T$	$E \rightarrow T$
$T + E$	$E \rightarrow T + E$
E	

General idea of bottom-up parsing

Syntax analysis

- Bottom-up parsing starts from the input token stream (whereas top-down starts from the grammar start symbol)
- It ***reduces*** a string to the start symbol by ***inverting productions***
    - trying to find a production matching the ***right hand side***

$E \rightarrow T + E \mid T$

$T \rightarrow \texttt{int} \times T \mid \texttt{int} \mid \varepsilon$

$E \leftarrow T + E \mid T$

$T \leftarrow \texttt{int} \times T \mid \texttt{int} \mid \varepsilon$

Why did we not immediately reduce the first "int" in the token stream using the production T → int?

$\texttt{int} \times \texttt{int} + \texttt{int}$	$T \rightarrow \texttt{int}$
$\texttt{int} \times T + \texttt{int}$	$T \rightarrow \texttt{int} \times T$
$T + \texttt{int}$	$T \rightarrow \texttt{int}$
$T + T$	$E \rightarrow T$
$T + E$	$E \rightarrow T + E$
E	

- Reading the productions in reverse (from ***bottom*** to ***top***) gives a ***rightmost derivation***

Taking right decisions: LR parsing

Idea: we extend the general idea of bottom-up parsing:

- Add the `EOF` token (`$`) and an extra start rule
  - This helps to uniquely identify when we constructed the root node of the parse tree
- As before, the parser uses a stack of terminal and NT symbols
- The parser looks at the current input token and decides between one of the following actions:
  - **shift**: Push the input token onto the stack, read the next token
  - **reduce**: Match the top symbols on the stack with a production right-hand side. Pop those symbols and push the left-hand side nonterminal. At the same time, build this part of the tree
  - **accept**: when the parser is about to shift `$`, the parse is complete
- The parser uses a DFA (encoded in a table) to decide which action to take and which state to **go to** after each shift action

LR parsing example

Notation: Stack ↑ Input

Grammar:

0	*S*	→	*Stmt* $
1	*Stmt*	→	id "=" *Exp*
2	*Exp*	→	id
3	*Exp*	→	*Exp* "+" id

```
↑ id = id + id $
```

shift

```
id ↑ = id + id $
```

shift

```
id = ↑ id + id $
```

shift

```
id = id ↑ + id $
```

reduce *Exp* → id

```
id = Exp ↑ + id $
     |
     id
```

shift

```
id = Exp + ↑ id $
     |
     id
```

shift

```
id = Exp + id ↑ $
     |
     id
```

reduce *Exp* → *Exp* "+" id

```
id = Exp ↑ $
     /|\
  Exp + id
   |
   id
```

reduce *Stmt* → id "=" *Exp*

```
  Stmt ↑ $
  /|\
id = Exp
     /|\
  Exp + id
   |
   id
```

LR(1) items

Syntax analysis

- The parser uses a DFA (a deterministic finite automaton) to decide whether to shift or reduce
  - The **states** in the DFA are **sets of LR items**

LR(1) item: $\boxed{X \to \alpha \bullet \beta \quad \texttt{t,s}}$

- An **LR(1) item** is a production extended with:
  - A **dot** (•), corresponding to the position in the input sentence
  - One or more possible **lookahead** terminal symbols, **`t,s`** (we will use **`?`** when the lookahead doesn't matter)
- The LR(1) item corresponds to a state where:
  - The topmost part of the stack is **α**
  - The first part of the remaining input is expected to match **β(t|s)**

Parser DFA: constructing state 1

First, take the start production and place the dot in the beginning…

Grammar:

```
0  S → E $
1  E → T "+" E
2  E → T
3  T → id
```

```
S → • E $              ?
```

Note that there is a nonterminal *E* right after the dot, and it is followed by a terminal $. Add the productions for *E*, with $ as the lookahead

```
S → • E $              ?
E → • T "+" E          $
E → • T                $
```

There is a nonterminal *T* right after the dot, which is followed by either "+" or $. Add the productions for *T*, with "+" and $ as the lookahead.
(We write them on the same line as a shorthand.)

state 1

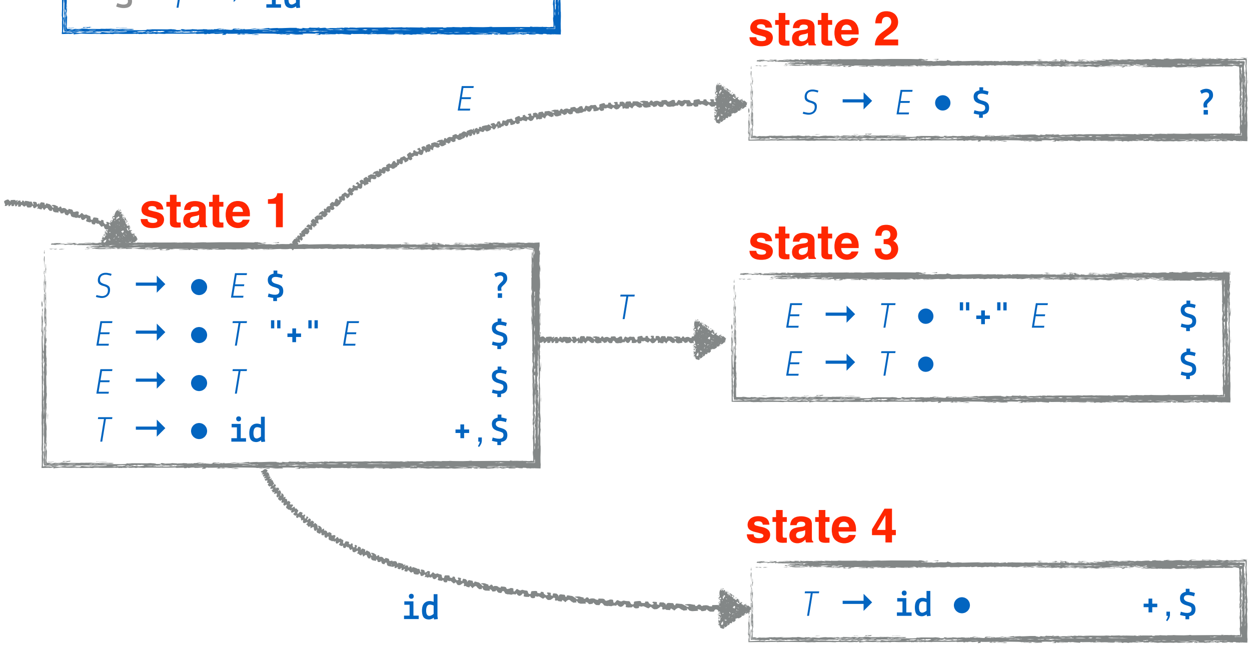
```
S → • E $              ?
E → • T "+" E          $
E → • T                $
T → • id             +,$
```

We have already added productions for all NTs that are right after the dot. Nothing more can be added. We are finished constructing **state 1**

Adding new productions for nonterminals following the dot, until no more productions can be added, is called **taking the closure** of the LR item set

Constructing the next states

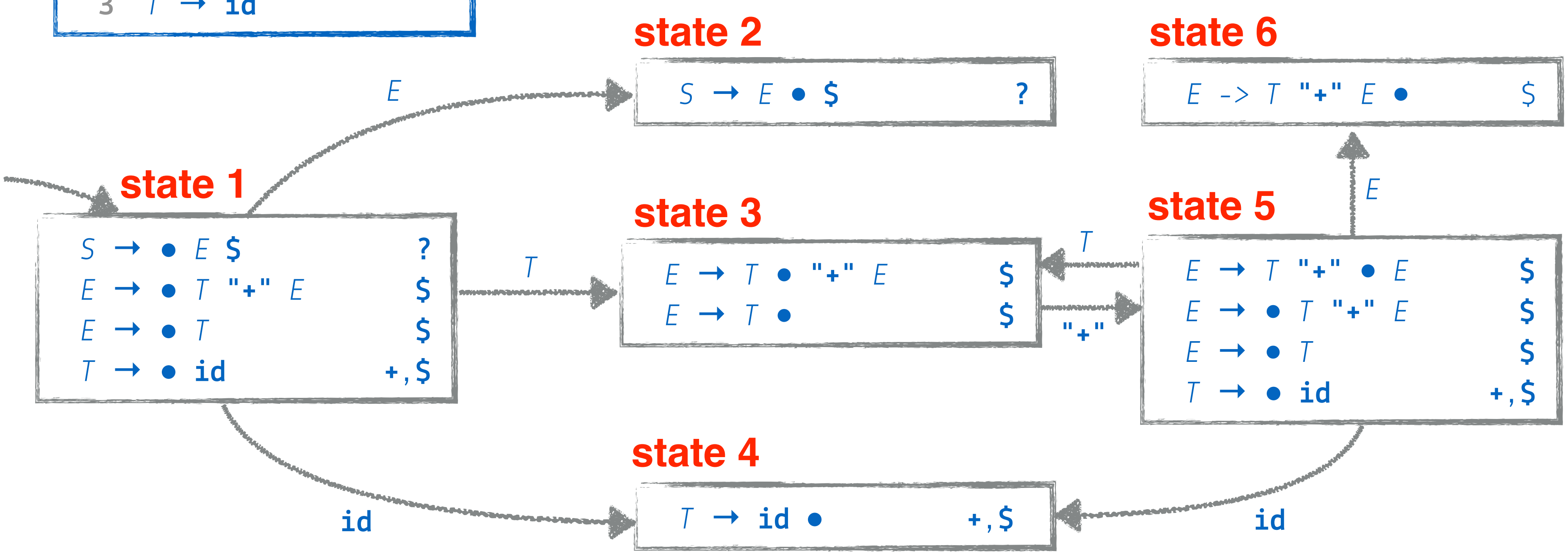
```
0  S → E $
1  E → T "+" E
2  E → T
3  T → id
```



Note that the dot is followed by *E*, *T*, and **id** in state 1.
For each of these symbols, create a new set of LR items, by advancing the dot past that symbol. Then complete the states by taking the closure
(Nothing had to be added for these states)

Completing the LR DFA


```
0  S → E $
1  E → T "+" E
2  E → T
3  T → id
```



Complete the DFA by advancing the dot, creating new states, completing them by taking the closure.

If there is already a state with the same items, we use that state instead.

Constructing the LR table

Syntax analysis

To write (or generate) a parser, we express the DFA using a table:

- For each token edge t, from state j to state k, add a **shift** action "**s k**" (shift and goto state k) to table[j,t]
  - = reading a token and pushing it onto the stack
- For each state j that contains an LR item where the dot is at the end, add a **reduce** action "**r p**" (reduce p) to table[j,t], where p is the production and t is the lookahead token
  - = popping the right-hand side of a production off the stack

state	"+"	id	$	*E*	*T*
1					
2					
3					
4					
5					
6					

- For each nonterminal edge X, from state j to state k, add a **goto** action "**g k**" (goto state k) to table[j,X]
  - = pushing the left-hand side nonterminal onto the stack

- For a state j containing an LR item with the dot to the left of $, add an **accept** action "**a**" to table[j,$].
  - If we are about to shift $, the parse has succeeded

Constructing the LR table

To write (or generate) a parser, we express the DFA using a table:

- For each token edge t, from state j to state k, add a **shift** action "**s k**" (shift and goto state k) to table[j,t]
  - = reading a token and pushing it onto the stack
- For each state j that contains an LR item where the dot is at the end, add a **reduce** action "**r p**" (reduce p) to table[j,t], where p is the production and t is the lookahead token
  - = popping the right-hand side of a production off the stack

state	"+"	id	$	*E*	*T*
1		s 4		g 2	g 3
2			a		
3	s 5		r p2		
4	r p3		r p3		
5		s 4		g 6	g 3
6			r p1		

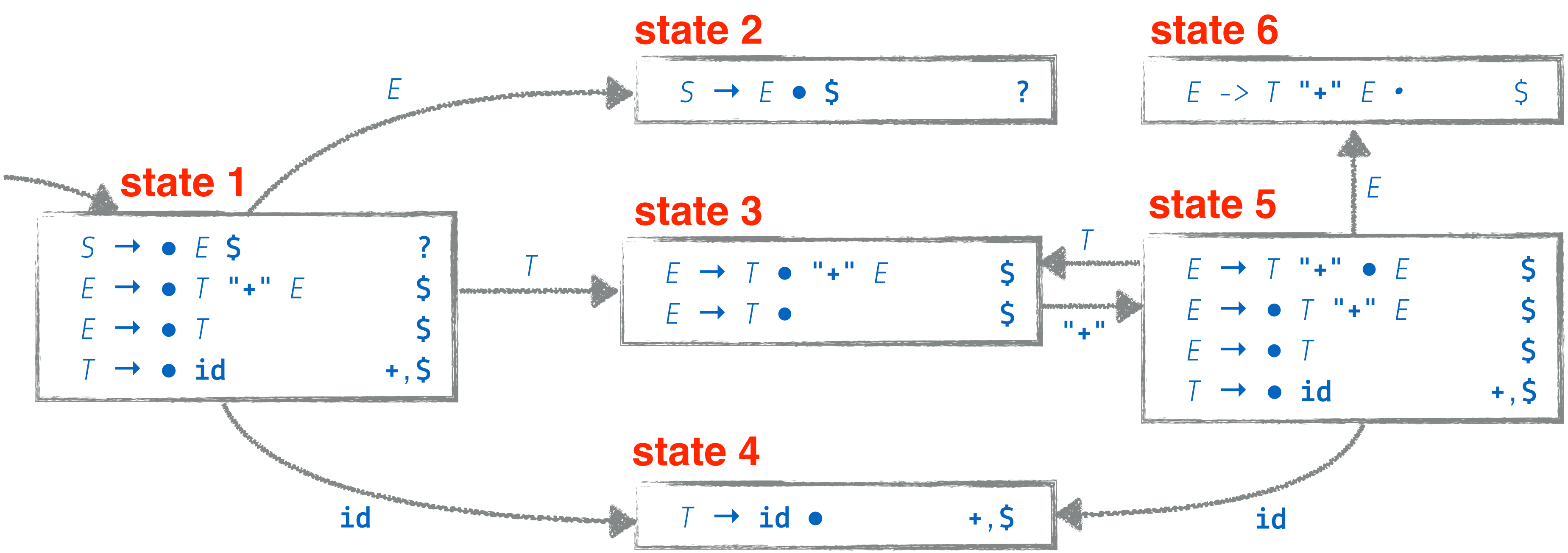
- For each nonterminal edge X, from state j to state k, add a **goto** action "**g k**" (goto state k) to table[j,X]
  = pushing the left-hand side nonterminal onto the stack

- For a state j containing an LR item with the dot to the left of $, add an **accept** action "**a**" to table[j,$].
  - If we are about to shift $, the parse has succeeded

DFA ↔ LR table

```
0  S → E $
1  E → T "+" E
2  E → T
3  T → id
```

state	"+"	id	$	*E*	*T*
1		s 4		g 2	g 3
2			a		
3	s 5		r p2		
4	r p3		r p3		
5		s 4		g 6	g 3
6			r p1		



LR parsing algorithm


```
push $;
push start state, s0;
word ← NextWord();

while (true) {
    state ← top of stack;
    if Action[state,word] = "reduce A → β" {
        pop 2 × |β| symbols;    // 2 × |β| due to terminal symbol and state
        state ← top of stack;
        push A;
        push Goto[state, A];
    }
    else if Action[state,word] = 'shift si' {
        push word;
        push si;
        word ← NextWord();
    }
    else if Action[state,word] = "accept" { break; }
    else Fail();
}
return success; /* executed break on 'accept' case */
```

Using the LR table for parsing

- Use a symbol stack and a state stack
  - The current state is the state stack top

- Push state 1 to the state stack

- Perform an action for each token:
- **Case Shift s:**
  - Push the token to the symbol stack
  - Push s to the state stack
  - The current state is now s
- **Case Reduce p:**
  - Pop symbols for the RHS of p
  - Push the LHS symbol X of p
  - Pop the same number of states
  - Let s1 = the top of the state stack
  - Let s2 = table[s1,X]
  - Push s2 to the state stack
  - The current state is now s2
- **Case Accept:** Report successful parse

state	"+"	id	$	*E*	*T*
1		s 4		g 2	g 3
2			a		
3	s 5		r p2		
4	r p3		r p3		
5		s 4		g 6	g 3
6			r p1		

LR parsing example

Syntax analysis

```
0  S → E $
1  E → T "+" E
2  E → T
3  T → id
```

state	"+"	id	$	E	T
1		s 4		g 2	g 3
2			a		
3	s 5		r p2		
4	r p3		r p3		
5		s 4		g 6	g 3
6			r p1		

Parsing `id "+" id $`

state stack	symbol stack	input	action
1		id "+" id $	shift 4
1 4	id	"+" id $	reduce p3
1 3	T	"+" id $	shift 5
1 3 5	T "+"	id $	shift 4
1 3 5 4	T "+" id	$	reduce p3
1 3 5 3	T "+" T	$	reduce p2
1 3 5 6	T "+" E	$	reduce p1
1 2	E	$	accept

Conflict in an LR table

Syntax analysis

```
0  S → E $
1  E → E "+" E
2  E → E "*" E
3  E → id
```

Different grammar to previous example!

Parts of the DFA:

state 3

```
E → E • "+" E        $
E → E "*" E •      "+"
```

"+"

state 5

Parts of the parse table:

state	…	"+"	…	…	…
…					
3					
…					

Fill in the parse table – what is the problem?

Conflict in an LR table


```
0  S → E $
1  E → E "+" E
2  E → E "*" E
3  E → id
```

Parts of the DFA:

state 3

```
E → E ● "+" E      $
E → E "*" E ●     "+"
```

"+"

state 5

Parts of the parse table:

state	…	"+"	…	…	…
…					
3		s 5, r p2			
…					

There is a shift-reduce conflict, the grammar is ambiguous.
In this case, we can resolve the conflict by selecting one of the actions.
To understand which one, think about what the top of the stack looks like. Think about what will happen later if we take the shift rule or the reduce rule.

Analyzing LR conflicts

Example parser generator output:

```
WARNING: resolved SHIFT/REDUCE conflict on [PLUS] by selecting SHIFT:
   REDUCE exp = exp PLUS exp
   SHIFT PLUS
Context:
   exp = exp PLUS exp • [PLUS]
   exp = exp • PLUS exp [PLUS]
```

Here, the parser generator automatically resolves the conflict by shifting.
Is this what we want?

"Context" lists the LR-items in the conflicting state

Align the dots in the state:

exp	→	*exp* **PLUS** *exp*	•			
exp	→		*exp*	•	**PLUS**	*exp*

The top of the stack might look like this:

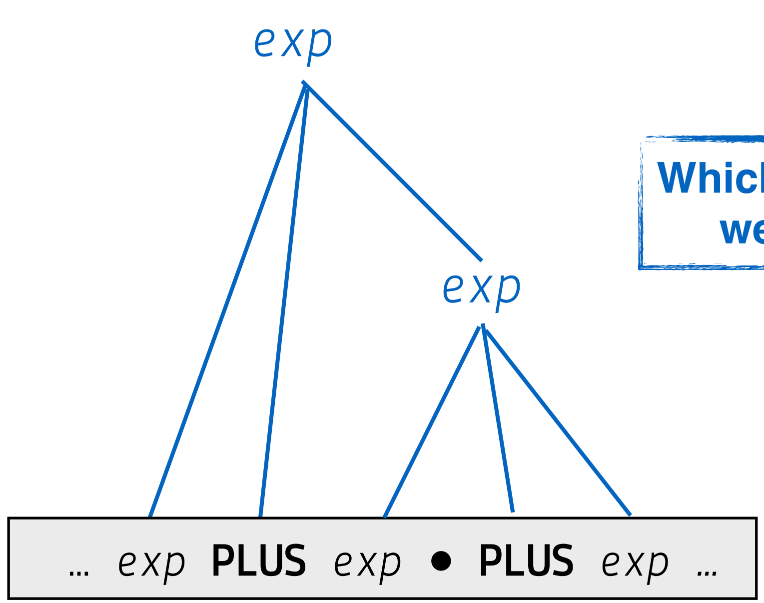
… *exp* **PLUS** *exp* • **PLUS** *exp* …

top of stack — **remaining input**

Analyzing LR conflicts

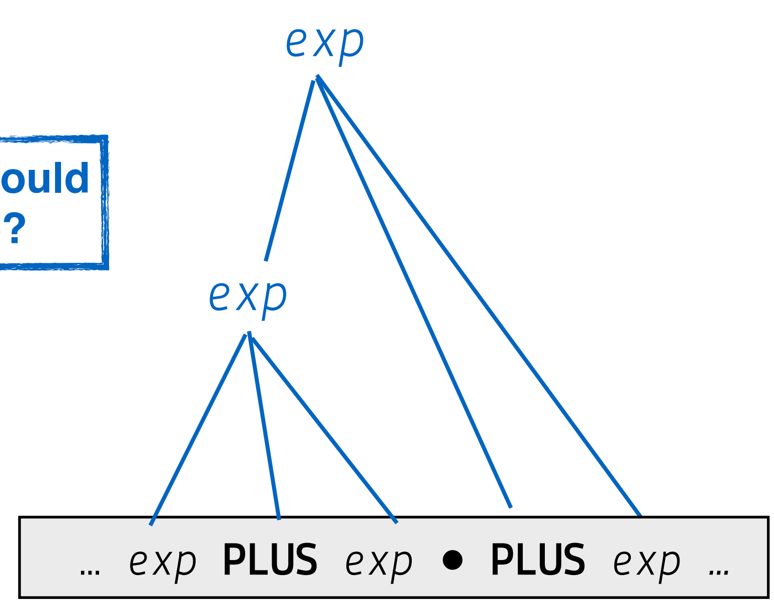
Align the dots in the state:

exp → *exp* **PLUS** *exp* •	**PLUS**
exp → *exp* • **PLUS** *exp*	**?**



if we shift

Which rule should we choose?



if we reduce

Different kinds of conflicts

$E \rightarrow E \bullet$ "+" E	?
$E \rightarrow E$ "*" $E \bullet$	"+"

a **shift-reduce** conflict

$A \rightarrow B\ C \bullet$	t
$D \rightarrow C \bullet$	t

a **reduce-reduce** conflict

Shift-reduce conflicts can sometimes be solved with precedence rules. In particular for binary expressions with priority and associativity.

For other cases, you need to carefully analyze the shift-reduce conflicts to see if precedence rules are applicable, or if you need to change the grammar.

For reduce-reduce conflicts, it is advisable to think through the problems, and change the grammar.

Typical precedence rules

Syntax analysis

Precedence rules for an LR expression grammar might look like this:

```
E -> E "==" E
E -> E "**" E
E -> E "*" E
E -> E "/" E
E -> E "+" E
E -> E "-" E
E -> ID
E -> INT

// Precedence rules
%right POWER
%left TIMES, DIV
%left PLUS, MINUS
%nonassoc EQ
```

Shift-reduce conflicts can be automatically resolved using ***precedence rules***

Operators in the same rule have the same priority
- e.g., PLUS, MINUS

Operators in an earlier rule have higher priority
- e.g. TIMES has higher priority than PLUS)

How the precedence rules work

Syntax analysis

A rule is given the priority and associativity of its rightmost token

For two conflicting rules with different priority, the rule with the highest priority is chosen:

$E \rightarrow E \bullet$ "+" E	?
$E \rightarrow E$ "*" $E \bullet$	"+"

reduce is chosen

$E \rightarrow E \bullet$ "*" E	?
$E \rightarrow E$ "*" $E \bullet$	"*"

shift is chosen

Two conflicting rules with the same priority have the same associativity

- Left-associativity favors reduce, right-associativity favors shift
- Non-associativity removes both rules from the table
    - input following that pattern will cause a parse error

$E \rightarrow E$ "+" $E \bullet$	"+"
$E \rightarrow E \bullet$ "+" E	?

reduce is chosen

$E \rightarrow E$ "**" $E \bullet$	"**"
$E \rightarrow E \bullet$ "+" E	?

shift is chosen

$E \rightarrow E$ "==" $E \bullet$	"=="
$E \rightarrow E \bullet$ "==" E	?

no rule is chosen

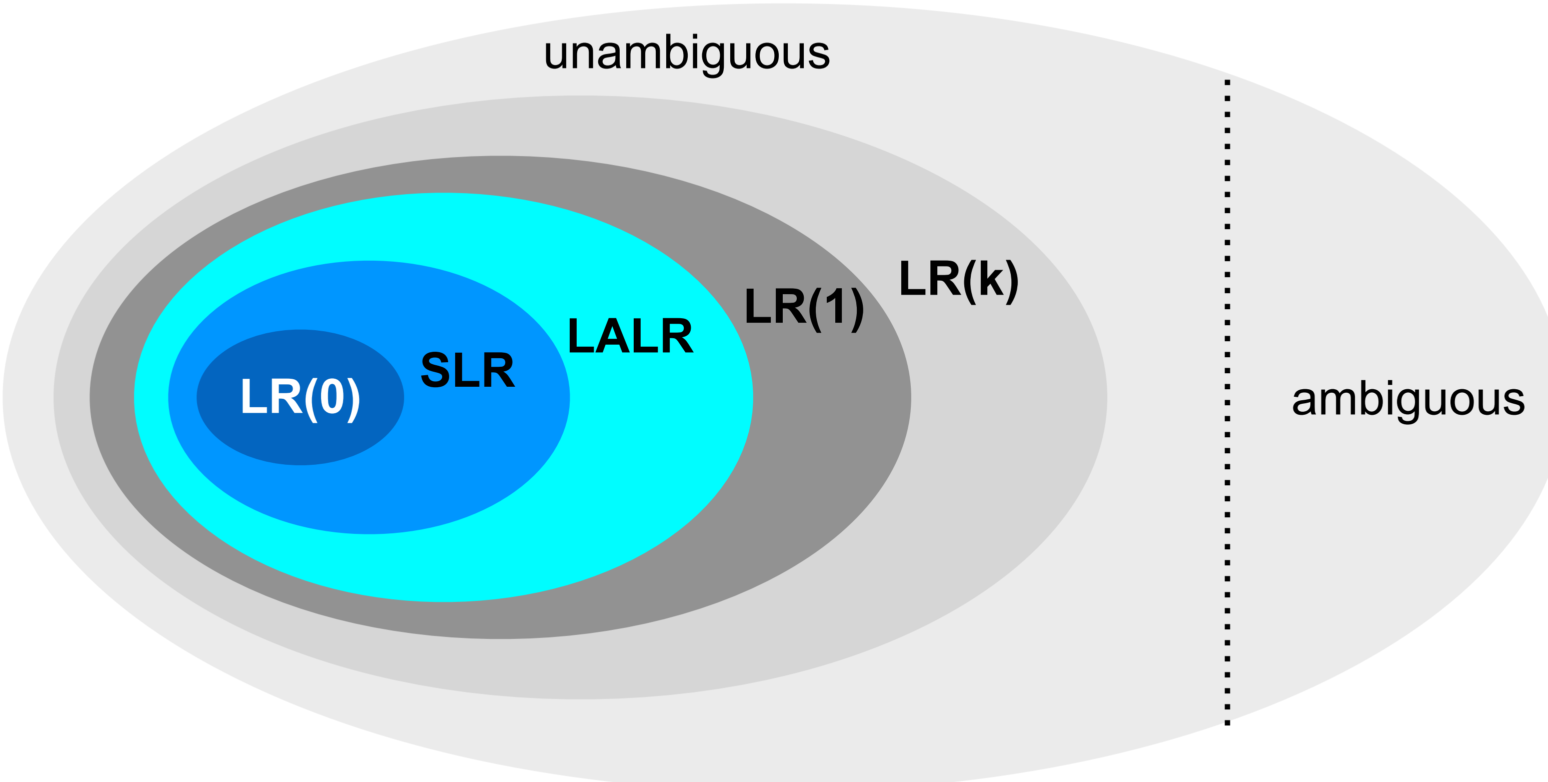
Different variants of LR(k) parsers

Type	Characteristics
LR(0)	LR items without lookahead Not very useful in practice
SLR Simple LR	Look at the FOLLOW set to decide where to put reduce actions Can parse some useful grammars
LALR(1) ***often used in practice***	Merges states that have the same LR items, but different lookaheads (LA) → leads to much smaller tables than LR(1) Used by most well known tools: yacc, CUP, Beaver, SableCC, ... Sufficient for most practical parsing problems
LR(1)	Slightly more powerful than LALR(1) Not used in practice – the tables become very large
LR(k)	Far too large tables for k>1

Different variants of LR(k) parsers

Syntax analysis

Different versions of LR(k) parsers can be used with grammars (= accept languages) of differing complexity:



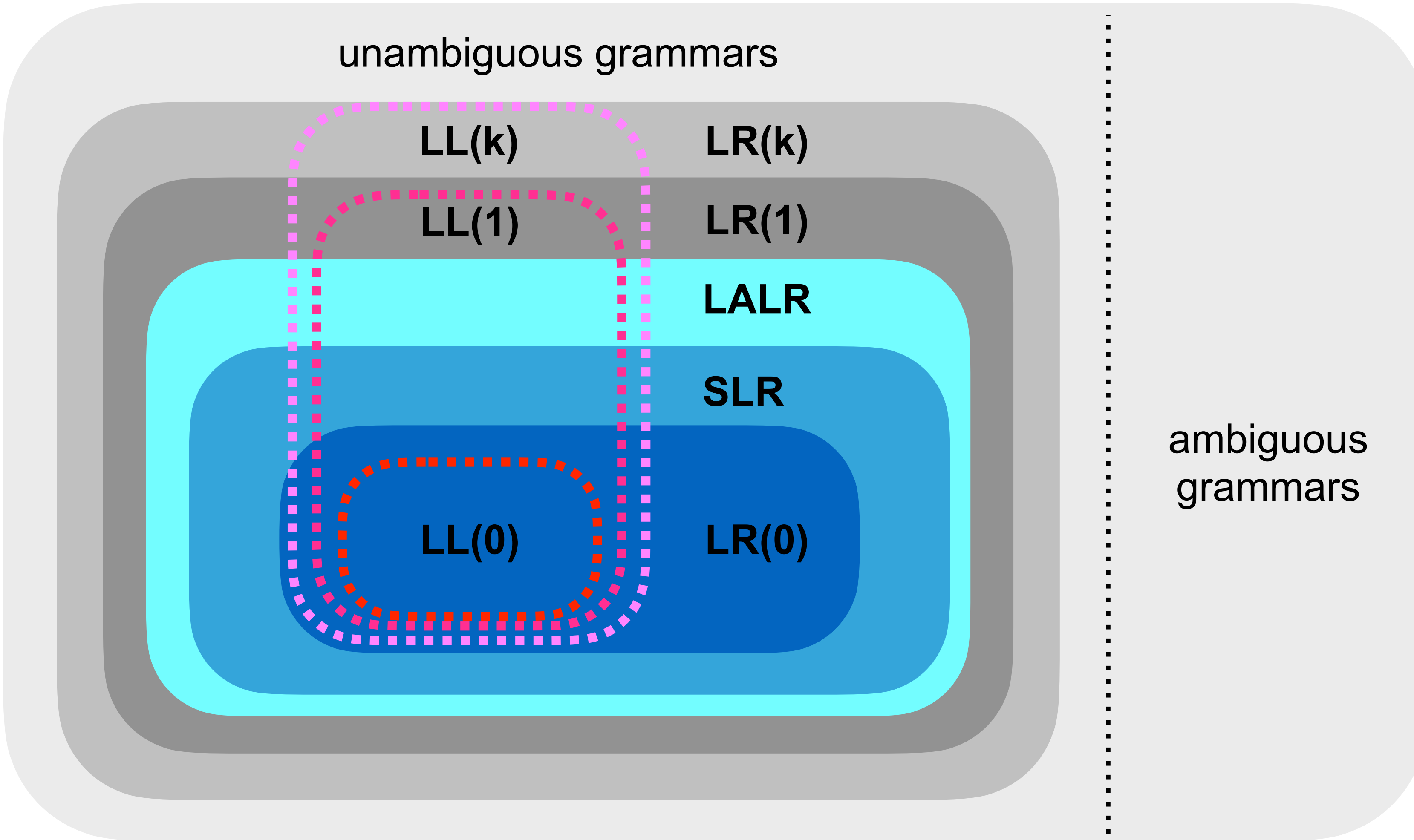
All context-free (type 2) grammars

LL(k) vs LR(k) parsers

	LL(k)	LR(k)
Parses input	Left-to-right	
Derivation	Leftmost	Rightmost
Lookahead	k symbols	
Build the tree	top down	bottom up
Select rule	after seeing the first k tokens	after seeing all of its tokens plus additional tokens
Left recursion	no	yes
Unlimited common prefix	no	yes
Resolve ambiguities through rule priority	dangling else	dangling else, associativity, priority
Error recovery	trial-and-error	good algorithms exist
Implement by hand?	possible	too complicated use a generator

LL(k) vs. LR(k) grammars

Syntax analysis



What's next?

- Practical considerations when constructing parsers
- Overview of Parser generators
- Using the yacc parser generator and examples
  - Interaction between yacc and lex